THE MINIATURIZATION OF THE AFIT RANDOM NOISE RADAR

THESIS

Aaron T. Myers, Captain, USAF

AFIT-ENG-13-M-37

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-37

THE MINIATURIZATION OF THE AFIT RANDOM NOISE RADAR

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Aaron T. Myers, B.S.E.E.

Captain, USAF

March 2013

AFIT-ENG-13-M-37

THE MINIATURIZATION OF THE AFIT RANDOM NOISE RADAR

Aaron T. Myers, B.S.E.E.
Captain, USAF

Approved:

[illegible]	21 FEB 2013
Peter J. Collins, PhD (Chairman)	Date
[illegible]	21 Feb 2013
John F. Raquet, PhD (Member)	Date
[illegible]	21 Feb 2013
Richard G. Cobb, PhD (Member)	Date

AFIT-ENG-13-M-37

Abstract

Advances in technology and signal processing techniques have opened the door to using an ultra-wideband random noise waveform for radar imaging. This unique, low probability of intercept waveform has piqued the interest of the United States Department of Defense (DoD) as well as law enforcement and intelligence agencies alike. Noise radar has shown tremendous potential in through the wall surveillance, monostatic and multistatic ranging, and communication. Ultimately, the Air Force Institute of Technology (AFIT) would like to explore the use of a random noise radar (RNR) as a potential unmanned aerial system (UAS) or remotely piloted aircraft (RPA) sensor.

As the employment of UASs and RPAs proliferates across the DoD, several design challenges must be considered: the ability to operate in noisy radio frequency (RF) environments (to include the presence of jamming), reliable and secure communication from the control facilities to the remote vehicles, and the option to operate surreptitiously. RNR systems have been shown to operate with all three of these desirable characteristics.

While AFIT's noise radar has made significant progress, the current architecture needs to be redesigned to meet the space constraints and power limitations of an aerial platform. This research effort is AFIT's first attempt at RNR miniaturization and centers on two primary objectives: 1) identifying a signal processor that is compact, energy efficient, and capable of performing the demanding signal processing routines and 2) developing a high-speed correlation algorithm that is suited for the target hardware. A correlation routine was chosen as the design goal because of its importance to the noise radar's ability to estimate the presence of a return signal. Furthermore, it is a computationally intensive process that was used to determine the feasibility of the processing component. To determine the performance of the proposed algorithm, results from simulation and experiments involving representative hardware were compared to the current system. Post-implementation reports of the field programmable gate array (FPGA)-based correlator indicated zero timing failures, less than a Watt of power consumption, and a 44% utilization of the Virtex-5's logic resources.

Dedicated to my youngest child.

Acknowledgments

First and foremost I would like to thank my Lord and Savior, Jesus Christ. This research effort has been a sobering reminder of how little I truly know, and it has been nice to know that He is in control and is there to provide us strength when needed the most. To my lovely wife...Thank you for your love and support throughout this program. Your patience and encouragement means the world to me. To my three kids, thank you for being there to greet me with hugs and excitement when coming home from too many long days at school. Your unconditional love fills my heart with joy. Finally, I could not have done this with out the guidance and encouragement from those at AFIT. Dr. Collins, the investment that you and others have put into educating me has taken a guy who couldn't spell "LO" and managed to turn me into a fairly competent engineer. To the rest of the "LO Mafia", thanks for your friendship and being willing to carry a scrub like me along for the ride.

Aaron T. Myers

Table of Contents

Page

Page

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
ADC	analog-to-digital converter
AFIT	Air Force Institute of Technology
ALU	arithmetic logic unit
CLB	configurable logic block
CORDIC	coordinate rotation digital computer
CPLD	complex programmable logic device
CS	compressive sensing
CW	continuous wave
DAC	digital-to-analog converter
DCR	direct correlation receiver
DFT	discrete Fourier transform
DIF	decimation in frequency
DIT	decimation in time
DSP	digital signal processor
DoD	Department of Defense
ECCM	electronic counter-counter measure
FCC	Federal Communications Commission
FFT	fast Fourier transform
FIR	finite impulse response
FPGA	field programmable gate array
GPU	graphical processing unit
HDL	hardware description language
IDE	integrated development environment

Acronym	Definition
IDFT	inverse discrete Fourier transform
IF	intermediate frequency
IFFT	inverse fast Fourier transform
IO	input/output
IP	intellectual property
LED	light-emitting diode
LFM	linear frequency modulation
LNA	low noise amplifiers
LPI	low probability of intercept
PCI	peripheral component interface
PLL	phase-locked loop
PNR	pseudo-random noise radar
PSD	power spectral density
PSU	Pennsylvania State University
R2^2SDF	radix-2^2 single-path delay feedback
RAM	random-access memory
RF	radio frequency
RIP	restricted isometric property
RMS	root mean squared
RNR	random noise radar
ROM	read-only memory
RPA	remotely piloted aircraft
RF	radio frequency
SMA	subminiature version A
SNR	signal to noise ratio

Acronym	Definition
SQNR	signal-to-quantization noise ratio
UART	universal asynchronous receiver/transmitter
UAS	unmanned aerial system
UNL	University of Nebraska, Lincoln
USB	universal serial bus
UWB	ultra-wideband
VCO	voltage-controlled oscillator
WGN	white Gaussian noise

THE MINIATURIZATION OF THE AFIT RANDOM NOISE RADAR

I. Introduction

Recent advances in technology and signal processing techniques have opened the door to using an ultra-wide band random noise waveform for radar imaging. This unique, low probability of intercept waveform has piqued the interest of the United States Department of Defense (DoD) as well as law enforcement and intelligence agencies alike. Noise radar has shown tremendous potential in through-the-wall surveillance, monostatic and multistatic ranging, and communication. Ultimately, Air Force Institute of Technology (AFIT) would like to explore the use of an random noise radar (RNR) as a potential unmanned aerial system (UAS) or remotely piloted aircraft (RPA) sensor.

As the employment of UASs and RPAs proliferates across the DoD, several design challenges must be considered: the ability to operate in noisy radio frequency (RF) environments (to include the presence of jamming), reliable and secure communication from the control facilities to the remote vehicles, and the option to operate surreptitiously. RNR systems have been shown to operate with all three of these desirable characteristics.

To integrate an RNR into a small UAS like the one pictured in Figure 1.1, a major question remains to be answered: can an RNR be designed small enough to meet the physical space constraints while simultaneously operating on very little power from the host vehicle? The current AFIT RNR configuration is simply too large for these types of airborne applications. As a result, new architectures will need to be explored for this sensor integration to become a reality. The quest for RNR miniaturization naturally leads to a series of questions:

Figure 1.1: Soldier Launching a Small UAS [5]

- What is a chip-based RNR architecture that would meet the requirements of the host vehicle?
- How does the miniaturized system perform when compared to the current AFIT RNR?
- Are there any performance compromises that must occur as a result of the miniaturization effort?
- Are the proposed architectures operationally viable?

1.1 Research Motivation

This research effort will serve as a concerted effort to utilize RNR for UAS collision avoidance and secure communication. The integration of an RNR into small RPAs or UASs would provide follow-on researchers with a covert sensor that could be used as a low probability of detection, jam resistant radar or communication device. A sensor of this type could prove useful in a swarm scenario as RNR signals behave well in the presence of interference.

1.2 Research Goals

AFIT's involvement in RNR design began in 2009. Since that time, researchers at AFIT have developed six monostatic nodes. With the exception of a new antenna design, the hardware configuration has remained relatively constant. Endeavors to miniaturize AFIT's RNR will take full advantage of the system understanding provided by past thesis work, but represent a drastic change in the hardware configuration and requisite signal processing algorithms.

Currently, AFIT's noise radar is a continuous wave (CW), direct correlation system. Radars of this type do not employ an intermediate frequency (IF) for signal transmission or reception. In other words, the CW noise signal is transmitted, received, and sampled at base-band. To determine a target's presence and estimate its range, the noise radar must execute a series of events. First, the AFIT noise radar directly samples the analog noise source with an ultra-high speed analog-to-digital converter (ADC) and stores the samples in memory. An array of filters are generated by digitally delaying copies of the transmission signal corresponding to a desired number of range bins. Finally, RF energy that is incident upon the receive antenna is sampled and compared to the transmission signal in each range bin. The comparison of the two signals is accomplished by calculating a statistical measure of similarity, correlation. When the two signals are highly correlated (i.e., the correlation coefficient is above a given threshold) a detection is declared.

Computing the correlation of two signals is a critical function required of the noise radar. However, generating correlation results is a computationally intensive process. Reducing the footprint and power consumption of the RNR while maintaining the ability to execute correlation routines in a timely manner presents a significant design challenge. For that reason, the primary focus of this document centers on the development of a high speed correlation routine that can be efficiently implemented in representative hardware. Consequently, this research effort can be summarized by the following goals:

1. Identify a preliminary RNR architecture that significantly reduces the size and power requirements of the current AFIT RNR.
2. Develop a correlation algorithm for the host architecture.
3. Develop a computer model of the proposed signal processing routine.
4. Demonstrate the algorithm in representative hardware to serve as a verification to the computer model.
5. Compare the new algorithm's performance to the current system.
6. Determine whether or not the proposed system is a good candidate for a small UAS or RPA.

A discussion of the fundamental principles and theoretical background necessary for accomplishing these research objectives is provided in Chapter 2. This theoretical basis will serve as the necessary foundation upon which the methodology of Chapter 3 will be built. The investigative steps outlined in Chapter 3 should provide the necessary information to conduct the analysis detailed in Chapter 4 and ultimately lead to an assessment of the proposed system. Finally, the conclusions drawn from data analysis are discussed in Chapter 5 and recommendations for future work will be contained therein. Now that the research goals have been proposed, a brief background will be provided to expose the reader to the foundations of noise radar systems and provide the appropriate context for this research effort.

1.3 Background

Noise modulated radar systems were first explored in the late 1950s and early 1960s. In 1959 Horton first proposed the use of noise-modulated systems as a distance measuring device which could serve as an altimeter for blind landings [15]. The concept of a random noise radar was later developed by researchers from Purdue University [12], and several prototype systems were developed in the United States and in the Netherlands [21, 32].

Soon after the first prototypes were built, research in the area of noise radar slowed significantly due to the computing power necessary to process ultra-wideband (UWB) signals. However, in the late 1990s researchers at the University of Nebraska and the Ohio State University began a resurgence in RNR exploration [24, 22, 35, 39]. Solid-state technology had finally advanced to the point where near real time processing of a wideband noise signal could be accomplished. Since that time, the intensity of RNR research and development has steadily increased. Commercially, RNR technology has been proposed as an anti-collision radar for vehicles, and applications for meteorological radar, marine radar, and air search have all been explored [13].

The use of a broadband stochastic signal to conduct detection and ranging is certainly a unique concept in a discipline that has existed since the 1930s. One obvious difference between RNRs and their more traditional brethren is that since radar's inception, an emphasis has been placed on the reduction of noise to improve target detection. Another more subtle difference is that the first radars operated at a single frequency or very narrow bandwidth of signals. RNR is a subset of an emerging class of UWB radars that are being developed. Both forms of radar have strengths and weaknesses. The theoretical discussion in Chapter 2 should bring more clarity to this classic engineering investigation.

AFIT's RNR design was largely based on that developed at the Pennsylvania State University (PSU) for through the wall surveillance. The researchers at PSU foresaw the need for a compact system and modeled their design on an architecture designed for a software defined radio [16]. Since Lai and Narayanan's paper was written in 2006, six working prototypes have been built at AFIT.

A closeup view of the actual hardware can be seen in Figure 1.2. Little was changed from the design developed by Lai et al. except that bandpass filters were added before the

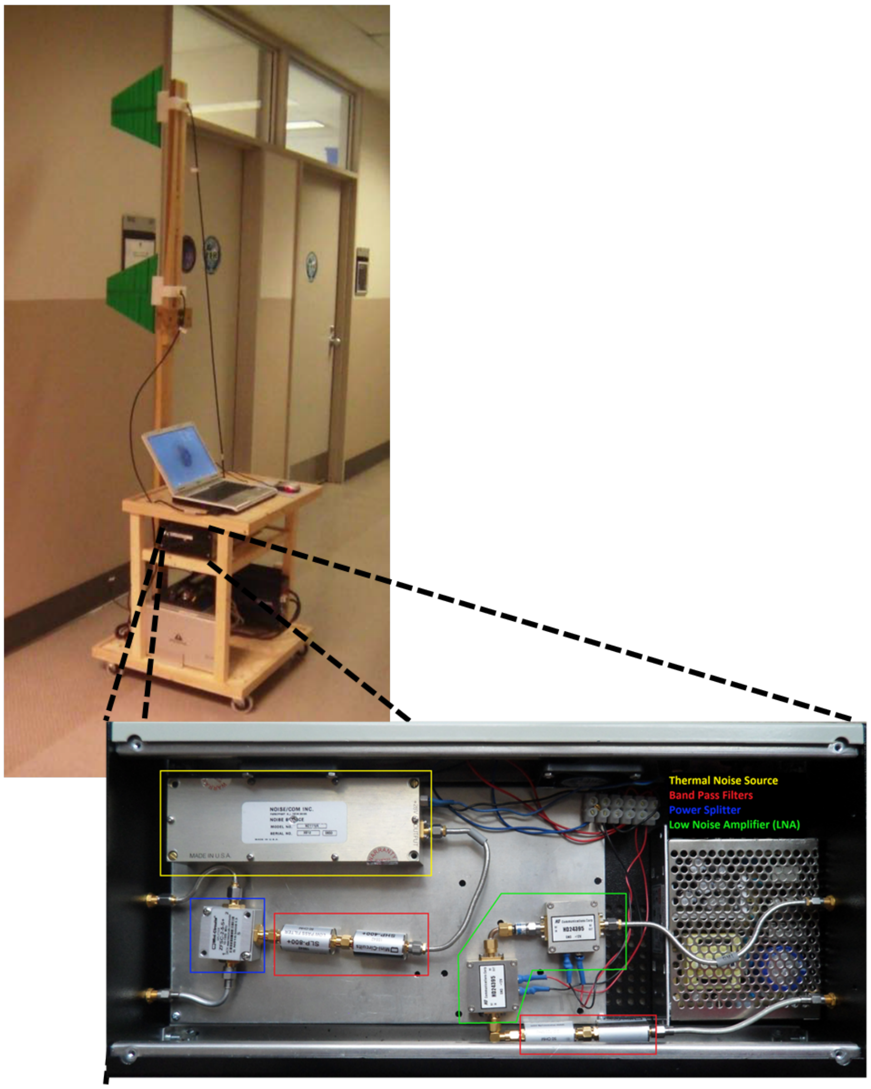


Figure 1.2: AFIT's RNR [6]

transmit antenna and after the receive antenna, shown highlighted in red in Figure 1.2. The component shown highlighted in yellow is the analog noise source that is used to generate the transmit waveform. The block with the blue surround is a power splitter that sends the noise signal to the ADC where it is sampled as the reference signal as well as to the transmit antenna to be radiated to the surrounding environment. Finally, the objects surrounded by green are the low noise amplifiers (LNA) used to amplify the return signal before being sent to the ADC and digital correlator. The ADC samples both the transmitted and received signals at 1.5 GHz. This data is sent to the laptop for signal processing in the MATLAB ® environment.

Much of AFIT's RNR research to date has centered on improving the computational efficiency of the current architecture. This will be the first attempt to make substantial changes to the current hardware configuration. Many of the lessons learned from past research will serve as a springboard for this effort. Of particular interest is the work that Priestly and Collins accomplished in pseudo-random noise radar (PNR) template replay. While the details of this effort will be discussed later, they were able to show that if implemented correctly, PNR template replay could maintain the low probability of intercept (LPI) nature of a truly random noise waveform [27, 6]. In addition to PNR template replay, the correlation process of the current AFIT RNR was often identified as a bottleneck in system throughput. As a result, Lievsay and Thorson proposed the development of graphical processing unit (GPU) or field programmable gate array (FPGA)-centric correlation routines that would increase the computational rate of the current system [18, 36].

1.4 Chapter Conclusion

This chapter defined the problem statement for this thesis and identified a few of the motivations for RNR miniaturization. The research goals were presented as a research blueprint. Finally, a background section was provided to capture the evolution of AFIT's RNR research. The next chapter presents the fundamentals of RNR operation and serves as a theoretical basis for RNR miniaturization.

II. Theory

2.1 Chapter Overview

This chapter presents the theoretical background necessary to explore the miniaturization of the AFIT's RNR. A two part introduction to RNR will begin with a mathematical description of a random noise waveform, description of the advantages stochastic signals provide to detection and ranging, and identification of several challenges associated with UWB systems. The RNR introduction will conclude with a theoretical basis for random signal generation and reception. Finally, the chapter will conclude by delving into specific concepts related to the miniaturization of RNR hardware.

2.2 Random Noise Radar

2.2.1 Random Noise Waveform.

To understand the operation of noise radar one must first look at the waveform. Many conventional radar systems employ a deterministic transmit signal. As a result, a clear mathematical representation can be derived. However, the stochastic nature of RNR waveform means that it can be defined only in terms of its statistics. UWB noise signals are often modeled as band-limited white Gaussian noise (WGN). Applying the WGN model, the UWB noise signals can be described using the following properties:

- The power spectral density of the UWB noise waveform is uniform and distributed evenly across the all frequencies.
- The amplitude of the noise signal is distributed according to the Gaussian probability density function with a mean of zero.
- The autocorrelation, $R_{xx}(\tau)$ is approximately an impulse at $\tau = 0$.

Adhering to the statistics above, a model can be developed to represent the time-frequency characteristics of the noise waveform. The first mathematical representation is

$$s(t) = s_I(t)\cos(\omega_o t) - s_Q(t)\sin(\omega_0 t), \tag{2.1}$$

where $s_I(t)$ and $s_Q(t)$ are zero-mean, Gaussian random variables. Alternatively, Equation (2.1) can be rewritten as

$$s(t) = a(t)\cos[\omega_o t + \phi(t)], \tag{2.2}$$

where $a(t)$ is a Rayleigh distributed amplitude and $\phi(t)$ is a uniformly distributed phase term [40].

2.2.2 Advantages of Ultra-Wide Band Noise Radar.

One of the main advantages that an UWB RNR has over traditional radar systems is range resolution. Because the random noise waveform will be uncorrelated with any other signal except for itself, the random noise waveform has an ideal "thumbtack" range-Doppler ambiguity function [17]. Theoretically, the range resolution, ΔR, of a radar system is determined by its time-bandwidth product. Range resolution can be found using the following equation:

$$\Delta R = \frac{c}{2B}, \tag{2.3}$$

where c is the speed of light (approximately 2.998×10^8 m/s) and B is the signal bandwidth [30]. When compared to traditional, narrowband radar systems, AFIT's RNR has an incredible range resolution. The transmission bandwidth of the AFIT noise radar gives it a range resolution of $\approx c/(2 * 400 \text{ MHz}) = 0.375$ m. Other prototype noise radars, such as the one developed at the University of Nebraska, Lincoln (UNL), have an instantaneous bandwidth of 1 GHz [23]. Inserting instantaneous bandwidth of UNL's RNR into Equation (2.3), down range distances on the order of 15 cm can be resolved.

In addition to improved range resolution, the random nature of an RNR's waveform gives the radar several favorable characteristics. The random noise waveform results in inherently low probability of intercept and low probability of detection, making it ideal for covert operations. Figure 2.1 shows a comparison of the ability to detect the presence of a noise radar compared to a conventional linear frequency modulation (LFM) radar.

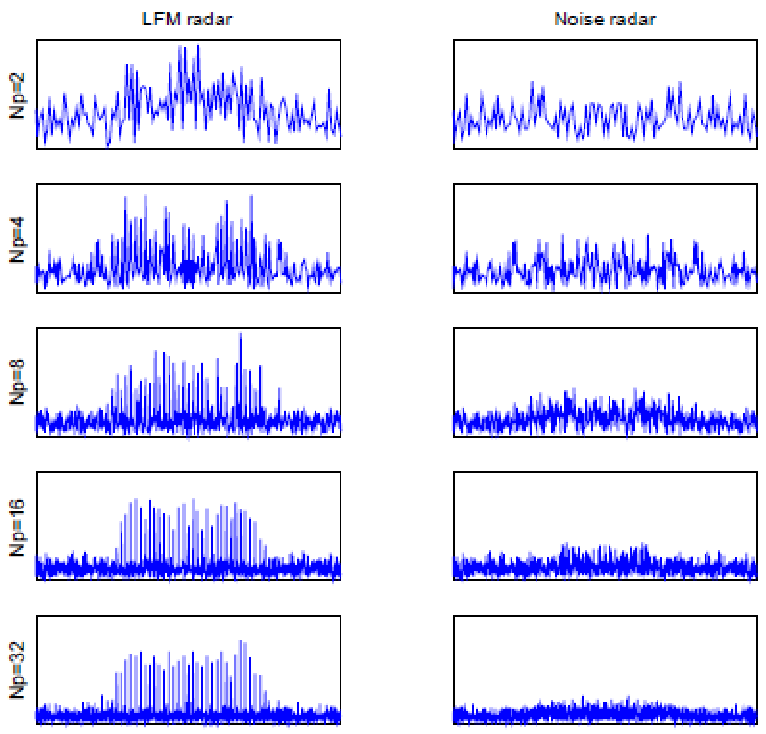


Figure 2.1: Fourier domain representation of an LFM radar (left) and RNR (right). Np is the number of radar pulses within the observation time. The SNR was 0 dB [34]

RNR's waveform allows it to operate in dense RF environments without causing interference to existing narrowband systems [10]. Figure 2.2 demonstrates that while the UWB signal stretches across the spectrum of the narrowband signal, the power level remains below the Federal Communications Commission (FCC) requirement for in-band interference. This characteristic has generated interest in the commercial sector as frequency allocation in the United States has become increasingly competitive and expensive. The DoD, on the other hand, is more interested in UWB RNR's capability as an electronic counter-counter measure (ECCM). Garmatyuk and Narayanan explored this issue and found that UWB RNRs perform much better in the presence of jamming than conventional LFM radars [10].

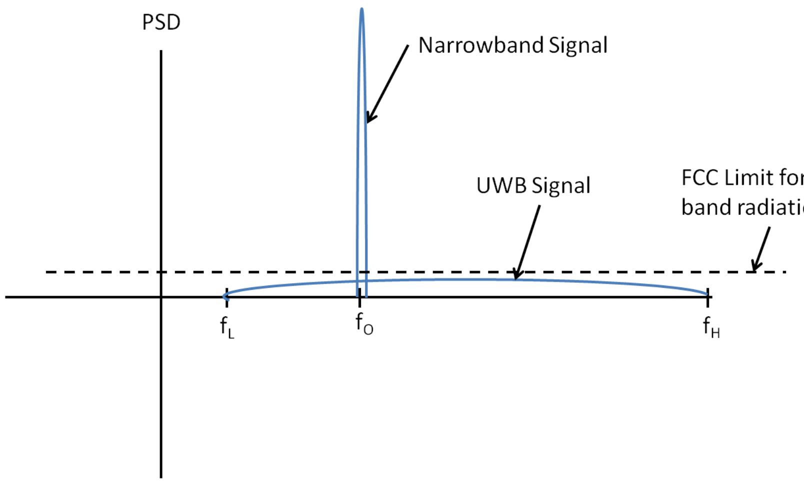


Figure 2.2: RFI from an UWB signal

Unlike conventional radars, RNRs have very simple architectures. This simplicity means that these radars are cost effective to produce. AFIT's noise radar has a predominately digital architecture making it tolerant of rapid reconfigures and updates.

2.2.3 UWB Noise Radar Challenges.

While UWB RNRs have inherent advantages, there are several weaknesses that they must overcome. One challenge is the ability to accomplish simultaneous velocity and ranging processing. Conventional Doppler processing requires phase coherency and narrowband signals. If these two conditions are satisfied, the Doppler shift can be found by:

$$f_d = \frac{2v}{\lambda} \cos\psi, \tag{2.4}$$

where λ is the wavelength of the center frequency, ψ is the angle between propagation of the radar's energy and the velocity of the target, and v is the velocity of the target [30].

To achieve phase coherence, many radar systems implement a heterodyne receiver to generate in-phase and quadrature information for both the transmitted and received waveforms. However, AFIT's noise radar implements a direct correlation receiver (DCR) to directly sample the outgoing and incoming waveforms. The random, non-repetitive nature of the CW transmit waveform leads to a lack of phase-incoherence [38]. Other RNR systems, such as the one proposed by Narayanan and Dawood, are heterodyne architectures capable of realizing phase coherency [23]. However, those systems give up the simplicity realized by DCR architectures by adding analog parts, increasing both size and weight. Furthermore, the presence of a carrier signal negates the low probability of detection characteristic enjoyed by baseband CW noise radars.

RNRs that are able to achieve phase coherence still have a problem that they have high fractional bandwidths. Equation (2.4) assumes the pulse transmitted by the radar is at a single frequency. AFIT's RNR transmits a CW noise signal with frequencies ranging from 350 to 750 MHz. This frequency range corresponds to wavelengths that are 37.5 to 75 cm. Significant error would be induced if the central wavelength was used to estimate target velocity [38]. In order to address these shortcomings, Lievsay and Thorson used time domain signal processing techniques to approximate the velocity of a target [18, 36]. They were able to utilize a technique that was proposed by Axelsson [1] that compressed the time scale of the transmitted signal by ΔT for each time sample where ΔT was given by [37]:

$$\Delta T = \frac{2v}{(c - v)f_s}. \tag{2.5}$$

This approach assumed that the target's velocity remained constant over the entire measurement window. Lievsay demonstrated a velocity resolution of 3 m/s, however, his technique required a lot of computing power and processing time [18]. Thorson lowered the processing time to approximately five minutes, but more work is required to accomplish real-time range and velocity processing [36].

Improving the transmit range of UWB RNRs is another challenge that must be overcome. The PSU noise radar (on which the AFIT RNR is based) has a transmit power of 23 dBm, an antenna gain of 6 dB, and a center frequency of 550 MHz with a bandwidth of 200 MHz [17]. The radar range equation given in [30] can be used to calculate the signal to noise ratio (SNR)

$$SNR = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4 L_s}. \tag{2.6}$$

Inserting the values for the PSU RNR above into Equation (2.6), the following expression is obtained:

$$SNR = \frac{(200\text{ mW})(6\text{ dB})^2(.545\text{ m})^2(1)}{(4\pi)^3 R^4}. \tag{2.7}$$

Equation (2.7) assumes a radar cross section, σ, of one and that the losses, L_s, are negligible. Table 2.1 provides SNR values calculated for three sample ranges.

Table 2.1: RNR SNR vs Range

Range	SNR
1 m	-33 dB
10 m	-73 dB
100 m	-113 dB

Although UWB RNRs perform well in low signal to noise conditions, distances approaching 100 m result in very poor SNRs making target detection difficult. UWB beamforming could be employed to improve the directional gain of the RNR antenna, thus improving RNR performance at long range.

2.3 Transmitter Theory

Noise radars are separated into two categories: continuous wave random noise and pseudo random noise. The distinction comes from the various ways of generating the transmit waveform. A description of both types of RNR and the underlying theory is presented below.

2.3.1 Continuous Wave Random Noise.

Continuous wave random noise radars derive their transmit waveform by continuously sampling and amplifying a signal from a truly random source. The AFIT RNR utilizes a solid-state thermal noise generator as its source. The noise source is a commercial off the shelf product designed to generate a band-limited noise waveform with characteristics described in Section 2.2.1. Thermal noise, often called Johnson noise, is a well studied topic as it is a source of contamination in many engineering disciplines. The root mean squared (RMS) voltage is given by

$$v_n = \sqrt{4k_B TRB} \tag{2.8}$$

where k_B is the Boltzmann constant (1.381×10^{-23} joules per kelvin), T is the absolute temperature in kelvin, R is the resistance in ohms, and B is the bandwidth [8]. Nelms was able to produce a plot of the power spectral density (PSD) for the AFIT noise source, shown in Figure 2.3. Throughout the AFIT RNR's operational bandwidth (350 MHz to 750 MHz) the noise source provided a nearly uniform response.

The benefit of continuous wave random noise radars is that they provide signals that are ideal for covert operation. The LPI nature of these signals makes them difficult to detect. Furthermore, continuous wave random noise signals are non-cyclical which prevents non-cooperative networks from intercepting and identifying characteristics of the transmitted waveform [27].

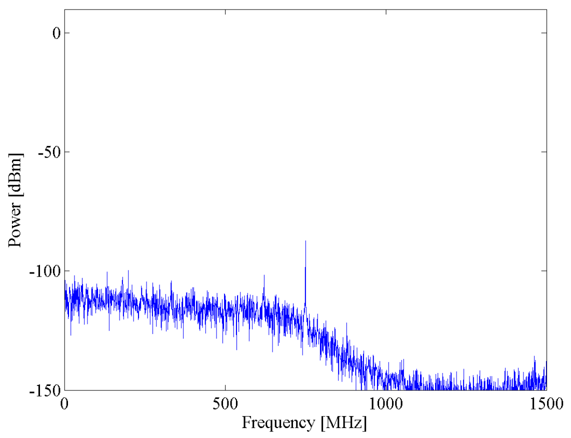


Figure 2.3: PSD of the AFIT RNR Noise Source [25]

While a continuous noise signal has its benefits, several challenges must be faced. First, constantly sampling noise sources with high-speed ADCs generates gigabytes of data and presents a computational challenge for even the most modern processors. Second, the noise sources for continuous wave RNRs are unique and must be sampled at the central node. This is an undesirable attribute if a distributed network is desired.

2.3.2 Pseudo Random Noise.

Pseudo random noise radars transmit templates of random numbers that have been stored in the system's memory. The templates are typically generated by one of two ways: storing snapshots of data collected from continuous random sources such as the thermal noise source describe above or by using random number generators. Both of these techniques have been explored at AFIT [6].

Pseudo random signals present researchers with more design options than continuous wave RNRs. Because the random templates are stored in memory, they can be distributed among cooperative networks. This simple change in the modus operandi allows RNR to become a true multi-static network for radar imaging. Furthermore, the orthogonality of one random template to another has enabled communication between nodes. In addition to design flexibility, Collins and Priestly demonstrated a significant improvement in the performance of distributive processing [6].

Although the UWB characteristics are preserved when transmitting pseudo random signals of sufficient length, the number of templates that can be stored on the host machine is limited by the available memory. The implication is that eventually any given template will be transmitted by the RNR more than once. If these repetitions occur regularly or in a predictable manner, detection by external networks become likely. One method to combat repetition in the transmitted signal is to develop a scheme where each radar or communication node is capable of synchronously generating the same set of random numbers. Then a cooperative network of pseudo random noise systems could generate new templates on the fly [6].

2.4 Receiver Theory

The receiver is the considered by many to be the most important component of a radar or communication system. The receiver not only accepts the incoming signals but is responsible for demodulation and hypothesis testing to determine whether or not the received waveform contains pertinent information. The fundamental concepts will be discussed in this section.

2.4.1 Sampling Theory.

Modern radars have processors that operate on digital signals. As a result, the analog waveforms incident upon the receiver must be digitized through the process of sampling. The discretization of analog signals leads to two questions: what sampling rate is sufficient for adequate representation of the continuous time signal and how many quantization levels are needed to capture the signal's amplitude?

To answer the first question the Nyquist sampling theorem must be considered. Assume that a continuous time signal, $x(t)$, has a band-limited Fourier transform, $X(f)$, that exists over the interval $[-B/2, B/2]$. Sampling $x(t)$ leads to the following expression:

$$\begin{aligned} x_s(t) &= x(t)\left\{\sum_{n=-\infty}^{\infty} \delta_D(t-nT_s)\right\} \\ &= \sum_{n=-\infty}^{\infty} x(nT_s)\,\delta_D(t-nT_s) \\ &= \sum_{n=-\infty}^{\infty} x[n]\,\delta_D(t-nT_s). \end{aligned} \tag{2.9}$$

The Fourier transform of (2.9) can be shown to be:

$$X_s(f) = \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X(f-kf_s)\,. \tag{2.10}$$

In other words, the Fourier transform of the discrete signal produces infinite copies of the continuous transform centered around integer multiples of the sampling frequency, f_s. If the sampling rate is too low, the spectrum of the original signal and the resulting copies would overlap making it impossible to recover $x(t)$. This observation led directly to the Nyquist sampling criterion. So long as f_s satisfies

$$f_s > B, \tag{2.11}$$

the spectrum of the original signal can be recovered by low-pass filtering $X_s(f)$ and multiplying by the sample period, T_s [30].

In addition to the sampling rate, discretization of the signal's amplitude, or quantization, must also be considered. This occurs because ADCs only have a finite number of bits to represent the amplitude. For example, a b-bit ADC has a range of $-2^{(b-1)}\,\Delta$ to $(2^{(b-1)}-1)\,\Delta$, where Δ is the step size (assumes that the amplitude is represented in two's-complement form). There is an inherent trade between the dynamic range of the ADC and quantization error [30]. The inclusion of a binary point (similar to the decimal point in base 10) provides a clearer illustration of this point. For every bit that the binary point moves to the right the dynamic range increases by a power of two ($\approx$ 6 dB). If bits are added to the right of the binary point, the precision of the fixed-point number is increased and quantization error is reduced. Richards and others were able to express the signal-to-quantization noise ratio (SQNR) as

$$SQNR\,(dB) = 6.02\,b - 10\,log_{10}\left(\frac{A_{sat}^2}{3\sigma^2}\right), \tag{2.12}$$

where b is the number of bits, A_{sat} is the largest value that can be represented without saturation and σ^2 is the power of the input signal [30]. If the input remains at a constant power and the dynamic range is normalized to A_{sat}, increasing the number of bits results in a 6 dB increase in the SQNR.

2.4.2 Discrete Fourier Transform.

Following the discussion on sampling, it seems appropriate to mention the transformation of temporal samples into the frequency domain. For continuous signals, the Fourier transform is defined as

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\, e^{-j\omega t}\, dt, \tag{2.13}$$

while the inverse Fourier transform is defined as [2]

$$x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)\, e^{j\omega t}\, d\omega. \tag{2.14}$$

Trying to use Equation (2.13) in practical systems is problematic in that $x(t)$ must be known for all time, t, before the transform can be calculated. Conversely, sampling $x(t)$ only provides a snapshot of the continuous time signal. Therefore, it is often assumed that the known portion of the signal can be extended periodically and, hence, the discrete Fourier transform (DFT) can be used to estimate the signal's spectral content. The equations for the DFT and inverse discrete Fourier transform (IDFT) are described by Rabiner and Gold as:

$$X_p[k] = \sum_{n=0}^{N-1} x_p[n]\, e^{-j(2\pi/N)nk}, \tag{2.15}$$

and

$$x_p[n] = \frac{1}{N}\sum_{n=0}^{N-1} X_p[k]\, e^{j(2\pi/N)nk} \tag{2.16}$$

respectfully, where p denotes periodicity [28].

The symmetry that exists between the DFT and IDFT is of practical importance. Algorithms that are developed to compute the DFT can be easily adapted for the IDFT. There will be more discussion on this matter in Section 3.4.

2.4.3 Fast Implementations of the Discrete Fourier Transform.

With spectrum analysis at the center of many operations in signal processing, researchers are continually looking for fast and efficient implementations of the DFT. These algorithms are often referred to as a fast Fourier transform (FFT).

One of the earliest and best known algorithms was introduced by James Cooley and John Tukey in 1965 [7]. If the number of elements in the transform, N, is chosen to be highly composite, then great efficiencies could be realized when computing the DFT. The smallest decomposition of N occurs when the number of elements in the transform is a power of two (i.e. $N = 2^v$) and forms the basis of the Radix-2 FFT. Equation (2.15) can be re-written as follows:

$$X[k] = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \tag{2.17}$$

where $W_N = e^{-j(2\pi/N)}$. Since the N-point sequence is even, it can be broken into two $N/2$-point sequences:

$$\begin{aligned} x_1[n] &= x[2n] \\ x_2[n] &= x[2n+1], \qquad n = 0, 1, 2, \ldots, \frac{N}{2} - 1. \end{aligned}$$

Equation (2.17) can then be rewritten as

$$X[k] = \sum_{n=0}^{N/2-1} x_1[n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x_2[n] W_{N/2}^{nk}, \tag{2.18}$$

effectively decomposing the N-point DFT into two N/2-point DFTs [28]. This decomposition can be repeated $log_2(N)$ until the computation is a two-point DFT. The two-point DFT is computed as follows [28]:

$$F[0] = f[0] + f[1]W^0 = f[0] + f[1] \tag{2.19}$$

$$F[1] = f[0] + f[1]W^{N/2} = f[0] - f[1]. \tag{2.20}$$

Graphically, the two point DFT is shown in Figure 2.4. The decomposition of the DFT

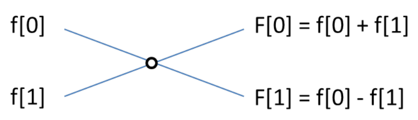


Figure 2.4: Two-point DFT (DIT Butterfly)

described above is often referred to as a decimation in time (DIT) FFT. Figure 2.5 provides a signal flow graph of the DIT FFT for an eight-point sequence.

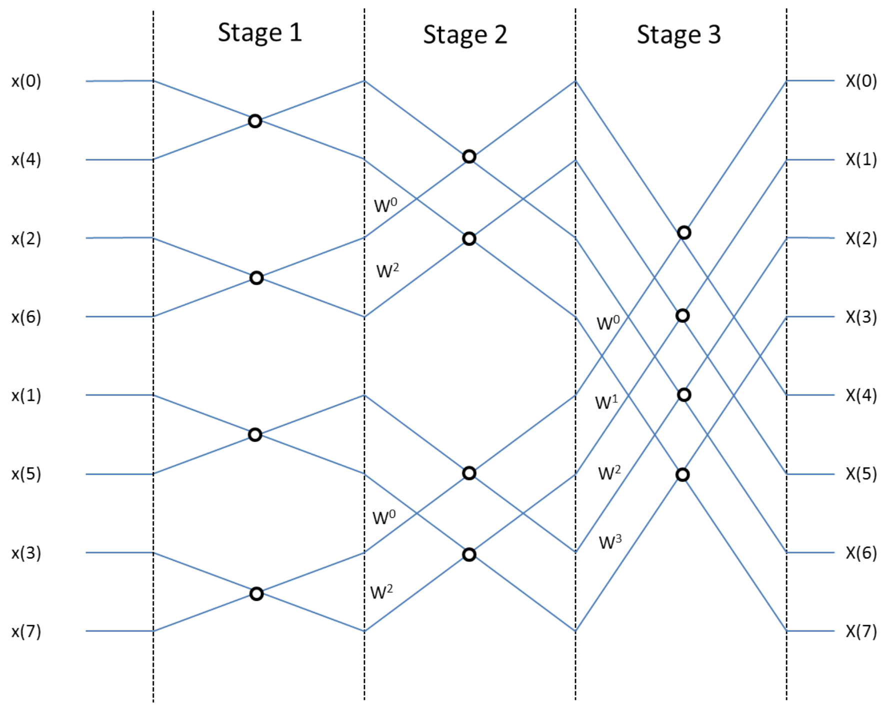


Figure 2.5: Signal Flow Graph for an 8-point DIT Radix-2 FFT

Another approach that is often taken is the decimation in frequency (DIF) algorithm. This algorithm starts by decomposing Equation (2.17) as follows [28]:

$$\begin{aligned} X[k] &= \sum_{n=0}^{N/2-1} x[n]W_N^{nk} + \sum_{n=N/2}^{N-1} x[n]W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x[n]W_N^{nk} + \sum_{n=0}^{N/2-1} x[n+N/2]W_N^{(n+N/2)k} \\ &= \sum_{n=0}^{N/2-1} \left[x[n] + e^{-j\pi k}x[n+N/2]\right] W_N^{nk}. \end{aligned} \tag{2.21}$$

Like the DIT algorithm, the decimation continues until a two-point DFT remains. For the DIF FFT, the two-point DFT is computed as follows [28]:

$$F[n] = x[n] + x[n+N/2] \tag{2.22}$$

$$G[n] = [x[n] - x[n+N/2]]W_N^n \qquad n = 0, 1, 2, \ldots, \frac{N}{2} - 1. \tag{2.23}$$

An illustration of the DIF butterfly is shown in Figure 2.6. An example of an eight-point

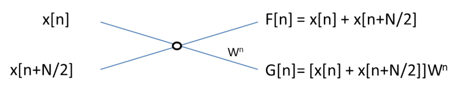


Figure 2.6: Two-point DFT (DIF Butterfly)

DFT utilizing the radix-2 DIF algorithm can be seen in Figure 2.7.

While the two algorithms produce similar flow graphs, the DIT algorithm requires that the input signal be in bit-reversed order and the DIF's input is in normal order. For this reason, the DIF algorithm is often preferred when designing hardware routines.

Since the Cooley-Tukey FFT was introduced in 1965, researchers have continued to look for ways to improve the computational efficiency of the DFT. This includes reducing the overall number of operations required to complete the transform while assessing how conducive it is to hardware implementation. In 1996, He and Torkelson

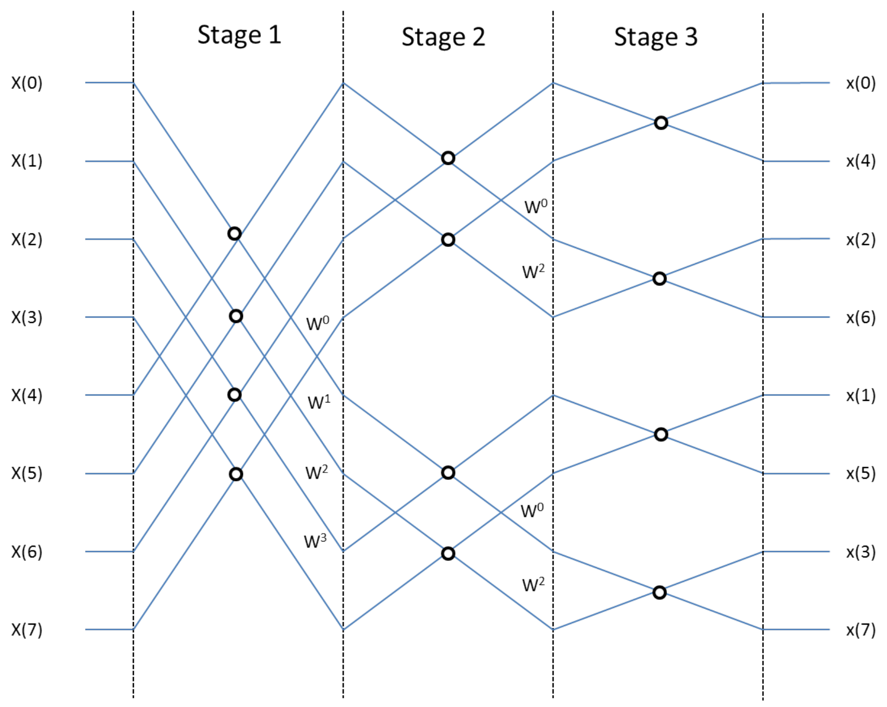


Figure 2.7: Signal Flow Graph for an 8-point Radix-2 DIF FFT

introduced the R2^2SDF DIF algorithm which has since gained traction in the FPGA design communities [14]. The algorithm's signal flow graph has spatial regularity, making it ideal for pipelining. Furthermore, the R2^2SDF has the added benefit of minimizing complex multipliers and memory registers while maintaining the basic structure and control as a radix-2 design [14].

The derivation of the R2^2SDF applies the following linear index map to Equation (2.17):

$$n =< \frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3 > N \tag{2.24}$$

$$k =< k_1 + 2k_2 + 4k_3 > N. \tag{2.25}$$

Following some simplification, He and Torkelson show that the result is four DFTs of length $N/4$

$$X[k_1 + 2k_2 + 4k_3] = \sum_{n=0}^{N/4-1} \left[H(k_1, k_2, n_3)\, W_N^{n_3(k_1+2k_2)} \right] W_{N/4}^{n_3k_3}, \tag{2.26}$$

where H is given by the expression [14]

$$H(k_1, k_2, n_3) = \underbrace{\overbrace{\left[x[n_3] + (-1)^{k_1} x[n_3 + N/2]\right]}^{\text{BFI}} + (-j)^{k_1+2k_2} \overbrace{\left[x[n_3 + N/4] + (-1)^{k_1} x[n_3 + 3N/4]\right]}^{\text{BFI}}}_{\text{BFII}}. \tag{2.27}$$

Recursive decomposition of a 16-point sequence results in the flow graph depicted if Figure 2.8.

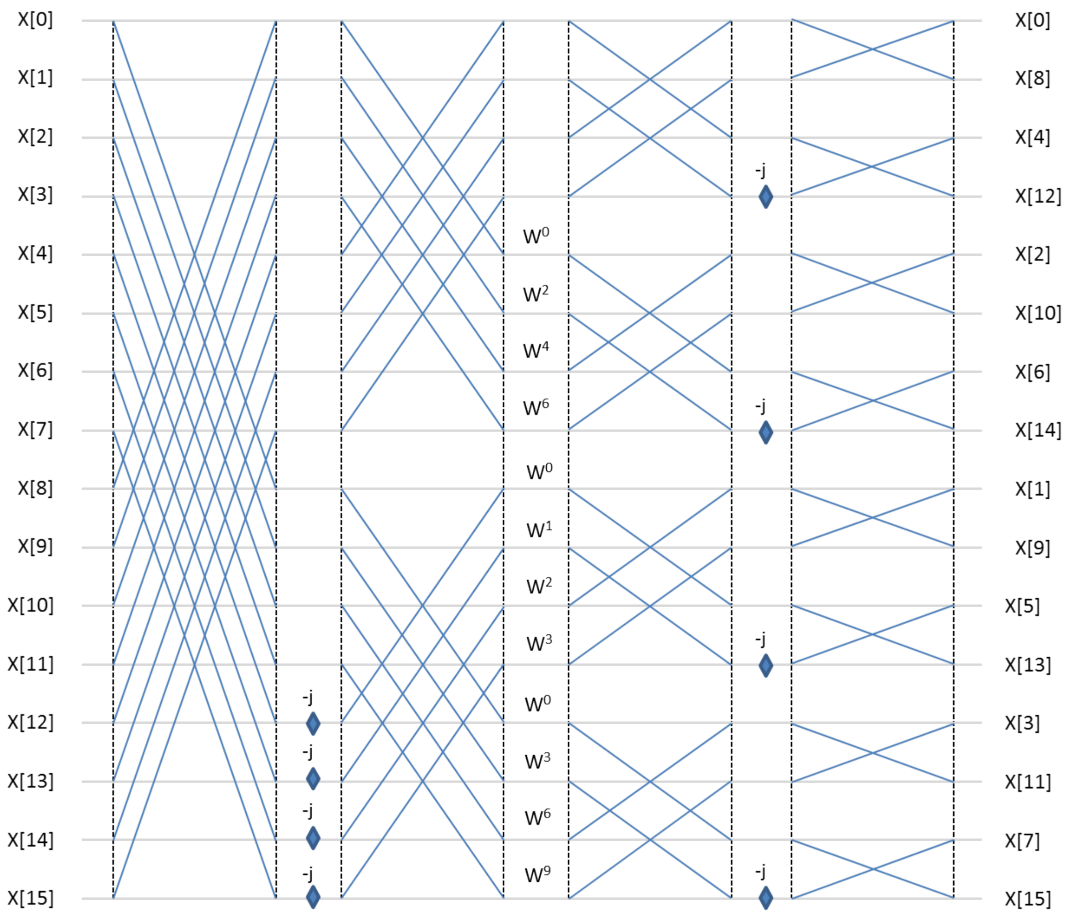


Figure 2.8: 16-Point Signal Flow Graph (Radix-2^2 SDF)

2.4.4 *Cross-correlation.*

Cross-correlation is a mathematical tool used in signal processing to determine the similarity of two waveforms and is often employed to detect a known signal in the presence of noise. In the case of two deterministic signals or stationary stochastic processes, x and y, the cross-correlation is defined by

$$c_{xy}[m] = \sum_{n=-\infty}^{\infty} x^*[n]\, y[n+m]. \tag{2.28}$$

Inspection of Equation (2.28) shows that the cross-correlation and convolution are closely related. Like the DFT, however, practical systems only provide a finite number of samples. Given the assumption made in Section 2.4.2, two periodic sequences $x_p[n]$ and $h_p[n]$ have transforms that can be described by Equation (2.15). This is an important result for linear-time-invariant systems as it allows for the development of circular convolution and correlation. For example, let $y_p[n]$ represent the circular convolution of $x_p[n]$ and $h_p[n]$:

$$y_p[n] = \sum_{l=0}^{N-1} x_p[l]\, h_p[n-l]. \tag{2.29}$$

The sequences $x[n]$ and $h[n]$ don't necessary have to be the same length, but they must be zero padded to the period of $y[n]$ for circular convolution to work [28]. An illustration is provided in Figure 2.9.

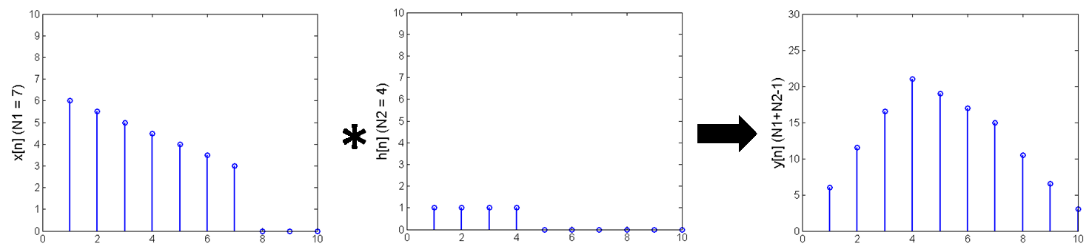


Figure 2.9: Example of Linear Convolution

Taking the DFT of $y_p[n]$ leads to the following result [28]:

$$\begin{aligned}
Y_p[k] &= \sum_{n=0}^{N-1}\left[\sum_{l=0}^{N-1} x_p[l]\, h_p[n-l]\right] e^{-j(2\pi/N)nk} \\
&= \sum_{l=0}^{N-1} x_p[l]\left[\sum_{n=0}^{N-1} h_p[n-l]\, e^{-j(2\pi/N)(n-l)k}\right] e^{-j(2\pi/N)lk} \\
&= H_p[k] \sum_{l=0}^{N-1} x_p[l]\, e^{-j(2\pi/N)lk} \\
&= H_p[k]\, X_p[k]. \qquad (2.30)
\end{aligned}$$

In other words, convolution and multiplication form a transform pair. A similar derivation can be done to show that the cross-correlation of two signals $x_p[n]$ and $y_p[n]$ can be calculated by taking the IDFT of a point-wise multiplication of $X_p[k]$ and $Y_p^*[k]$ [28]:

$$\begin{aligned}
\mathrm{DFT}^{-1}[X_p[k]\, Y_p^*[k]] &= \frac{1}{N}\sum_{k=0}^{N-1} X_p[k]\, Y_p^*[k]\, e^{j(2\pi/N)mk} \\
&= \frac{1}{N}\sum_{k=0}^{N-1}\left[\sum_{r=0}^{N-1} x_p[r]\, e^{-j(2\pi/N)rk}\right] \times \left[\sum_{s=0}^{N-1} y_p^*[s]\, e^{j(2\pi/N)sk}\right] e^{j(2\pi/N)mk} \\
&= \sum_{r=0}^{N-1}\sum_{s=0}^{N-1} x_p[r]\, y_p^*[s]\left[\frac{1}{N}\sum_{k=0}^{L-1} e^{j(2\pi/N)k(m-r+s)}\right] \\
&= \sum_{r=0}^{N-1}\sum_{s=0}^{N-1} x_p[r]\, y_p^*[s]\, \delta(m-r+s) \\
&= \sum_{s=0}^{N-1} y_p^*[s]\, x_p[m+s]. \qquad (2.31)
\end{aligned}$$

This result has greatly improved the computational efficiency of correlation algorithms, reducing the number of complex operations from N^2 to $N \log N$ [2].

2.4.5 Matched Filtering.

In radars and communication systems alike, performance and SNR are directly related. Therefore, removing extraneous noise from the received waveform while enhancing the signal of interest is paramount. In vector notation, the power of the signal component at the output of a finite impulse filter is shown by Richards and others as

$$|y|^2 = y^* y^T = \mathbf{H^H X^* X^T H}, \qquad (2.32)$$

where **X** is the signal of interest and **H** is the filter [30]. The expected value of filtered noise is

$$|y|^2_{\text{noise}} = \mathbf{H^H R_I H} \tag{2.33}$$

with $\mathbf{R_I}$ being the noise covariance matrix. From these two expressions, the SNR is simply a ratio of the signal and noise powers. The authors go on to find an **H** that maximizes the SNR. As a result of this optimization, the ideal filter is found to be

$$\mathbf{H} = \mathbf{kR_I^{-1}X}^*. \tag{2.34}$$

For the special case that the noise in the transmission channel is white and Gaussian, $\mathbf{R_I} = \sigma^{\mathbf{2}}_{\mathbf{n}}$. If k is chosen to be $1/\sigma^2$, the result is simply [30]

$$\mathbf{H} = \mathbf{X}^*. \tag{2.35}$$

In other words, the ideal filter in the presence of white noise is just the complex conjugate of the signal of interest. Hence, **H** is known as a “matched filter”.

2.5 Chapter Conclusion

This chapter presented the basic principles of noise radar technology. The next chapter will tie these principles to the research effort of miniaturizing the AFIT RNR architecture and corresponding algorithm development.

III. System Description and Methodology

3.1 Chapter Overview

The miniaturization of the AFIT RNR is a complex architectural design problem. To ensure that this design effort is accomplished in an efficient and logical manner, methodologies related to design, algorithm development, and system characterization will be discussed in this chapter. A preliminary set of requirements will be outlined in the following section in order to justify the selection of critical hardware components. The remainder of the chapter will center around developing signal processing routines that can be implemented on the host system and the evaluation of the resulting system's performance.

3.2 Requirements Definition

The design of a new architecture for AFIT's noise radar is a classic systems engineering problem. Figure 3.1 depicts the V-model often used to model processes associated with system design and implementation [26]. To begin the process, one must start by defining the concept of operation. This was accomplished in the thesis introduction. The next step shown in Figure 3.1 is identifying the system requirements.

The design requirements for a miniature RNR can be separated into two major categories: the requirements at the system of systems level (i.e., the requirements that originate from the interactions between the host vehicle and the RNR) and those at the system level (i.e., the attributes that enable the RNR to operate as intended). A complete understanding of how the RNR will be integrated into the host vehicle is beyond the scope of this research effort. However, before the second category can be addressed, a top-level description of the operating environment must be understood.

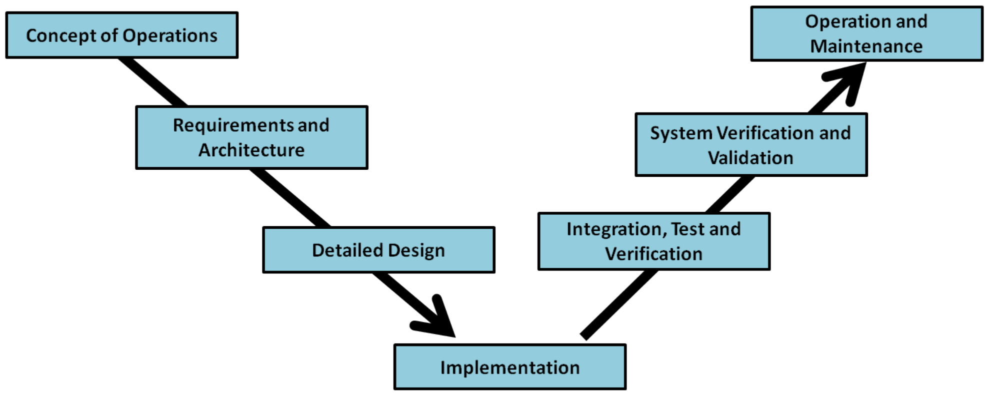

Concept of Operations
Requirements and Architecture
Detailed Design
Implementation
Integration, Test and Verification
System Verification and Validation
Operation and Maintenance

Figure 3.1: Systems Engineering V-Model

3.2.1 System of Systems Requirements.

Smaller UASs are highly constrained in cargo capacity and battery power. For an RNR to operate in this environment, it must be compact, lightweight, and consume minimal amounts of power. Finally, the input/output (IO) interface and communication protocol of the host vehicle will need to be defined before an RNR design can be finalized.

3.2.2 System Requirements.

In order to define component level requirements it is a good idea to begin with a description of the functional baseline. The basic functions of an RNR are outlined in Figure 3.2. These functions are shared by a majority of existing radar systems. The functional hierarchy is separated into two main paths: the transmit functions and the receive functions. In order for the radar to transmit a signal, it must generate the waveform, delay (or store) a copy of the generated waveform for coherence, and send the signal through the transmit hardware and out the antenna. When the radar is ready to execute the receive function, it must accept the incoming waveform through the receive hardware, sample the incoming signal, and compare the received signal to the delayed transmit waveform to determine whether or not it is a target return.

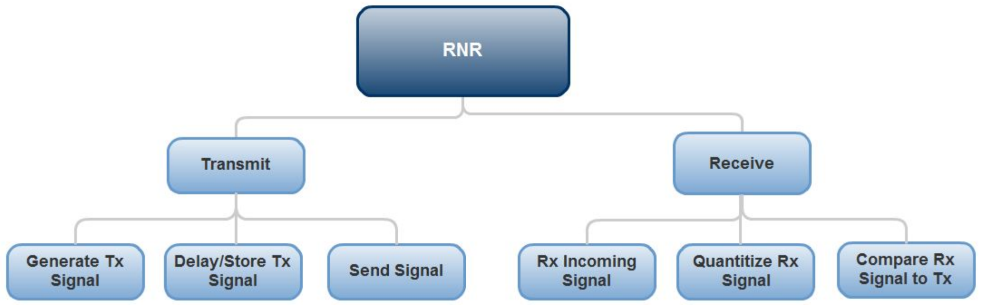


Figure 3.2: RNR Functional Hierarchy

It is strongly desired that the miniaturized RNR achieve a performance that rivals AFIT's current RNR configuration. As a result, the proposed system will need to process RF bandwidths of at least 750 MHz. Furthermore, the proposed design should provide two possible modes of operation: radar and communication (i.e., the received signal will need to be cross-correlated with either delayed versions of the transmit waveform or communication templates). A preliminary set of requirements for the miniature RNR is provided in Table 3.1.

Table 3.1: Miniature RNR Requirements

Specification	AFIT RNR	Miniature RNR	
		Threshold	Objective
Bandwidth	750 MHz	750 MHz	1 GHz
Computation Speed - 1024 pt Correlation	1 ms	10 µs	1 µs
Transmit Power	1 W	5 W	5 W
Size	9 ft^3	0.5 ft^3	0.2 ft^3
Power Consumption	110 W	20 W	10 W
ADC/DAC Resolution	8-bit	8-bit	12-bit

The development of a miniature RNR requires the successful completion of several component-level design tasks. These tasks have been summarized in Figure 3.3. In addition to enumeration, Figure 3.3 also captures the design time and level of effort required to complete each task. Given the time constraints placed on this research effort, the focus is centered on the design of the correlation algorithm and its implementation. Successful completion of this task will demonstrate the feasibility of a miniature RNR and clears a significant hurdle in system design.

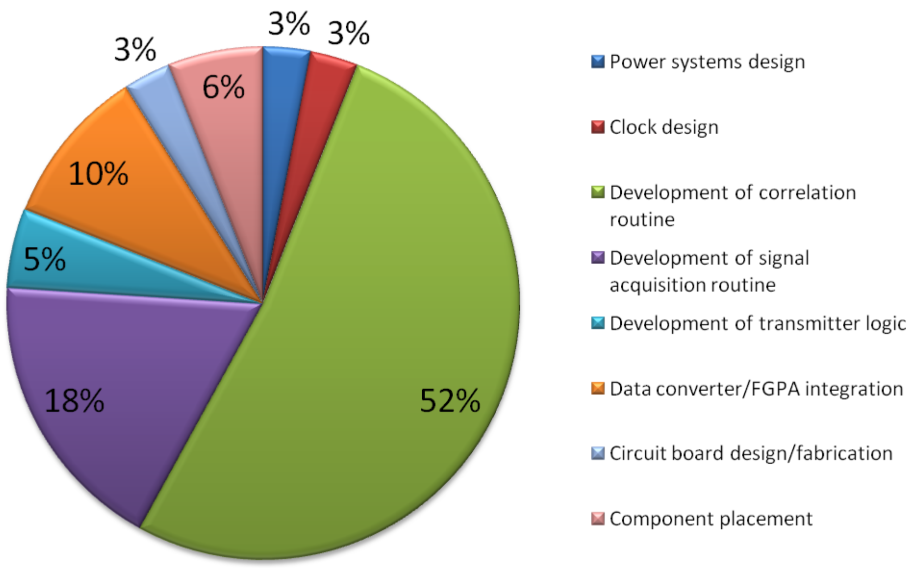


Figure 3.3: RNR Miniaturization Tasks

3.3 Hardware Design

With a basic set of requirements defined for the miniaturized RNR, some design choices need to be made before signal processing routines can be developed. While the development of a detailed architecture is left as a follow-on effort, the primary computational component will be identified. Furthermore, some design considerations for the RNR's peripheries will be outlined in this section.

3.3.1 Signal Processor.

The need to minimize size and power consumption while simultaneously maintaining the ability to accomplish high-performance signal processing routines narrows the field of possible signal processors down to two categories: digital signal processors (DSPs) and FPGAs. To choose between them a comparison of both architectures is needed.

DSPs have been around for many years and are considered a critical component of many electronic systems. DSPs are specially designed microprocessors that are well suited for arithmetic-intensive tasks. The algorithms written for DSPs are often programmed in C and are executed sequentially as each element has to pass through an arithmetic logic unit (ALU). To meet the ever increasing need for high-speed signal processing, modern DSPs have been designed with multiple ALU cores allowing designers to take advantage of parallel processing [44].

While DSPs have come a long way in computational throughput, they are still severely limited by clock speed and their sequential, instructional based architectures. Despite the availability of multicore DSPs, many designers have resorted to multiple DSP devices on a single board [44]. In the context of a miniature RNR, this would be counter productive to the goals of minimizing the overall size of the design and limiting power consumption. Additionally, multiple device or multicore designs often shift the focus of programmers from executing the signal processing routine to scheduling tasks and resources across the multiple devices. The result is a significant increase in code that functions as overhead and an exponential trend in system performance. In other words, it may take two devices to double the throughput of a signal DSP, but to double it again would require four devices [44].

The primary alternative to a DSP is an FPGA. While FPGAs vary by manufacturer and price, most contain the following elements: massive arrays of uniform configurable logic blocks (CLBs), memory, DSP slices, IO transceivers, and clock management devices. The FPGA's architecture enables two modules, A and B, to operate in parallel and independently. Designers are able to tailor implementations to match the system's requirements (i.e., high speed signal processing routines would utilize multiple channels and maximize parallelism while lower-rate designs could be designed to minimize resources reducing power consumption).

The delineation of the two may be clearly understood through an example. The one used by Zatrepalek in [44] is the finite impulse response (FIR) filter. The FIR filter was chosen because it is the most commonly used signal processing element, and it provides a good illustration of the strengths and weakness of the DSP and FPGA architectures [44].

Mathematically a simple FIR filter is represented by:

$$Y_n = \sum_{i=0}^{N-1} k_{n-1} S_i, \tag{3.1}$$

where S is a continuous stream of input samples, k_{n-1} are the filter coefficients, n is a particular instant in time, and Y is the filtered signal. The basic steps for implementing the FIR filter are: sample the incoming signal, organize the samples in a memory buffer, multiply each sample with the corresponding coefficient and accumulate the result, and output the filtered result [44].

Zatrepalek implemented a 31-tap multiply-and-accumulate FIR filter in a DSP running at a clock rate of 1.2 GHz. The maximum performance was measured at 9.68 MHz, or 9.68 Megasamples per second (MS/s). On the other hand, a parallel implementation of the filter on an FPGA could output a result on every clock cycle while simultaneously leaving a majority of the chip's resources to execute other processes or algorithms. The maximum performance that could be achieved on a Virtex 7 FPGA was calculated to be 600 MHz or 600 MS/s [44].

The efficiencies and computational performance of an FPGA make it the ideal choice for the miniaturized RNR. In addition to handling the data rate and high-speed correlation associated with a baseband receiver, the FPGA could potentially execute the transmit and receive functions simultaneously. Furthermore, an FPGA would provide follow-on researchers the ability to quickly reconfigure the design without changing or rewiring any hardware.

3.3.2 Data Converters.

The ability to convert an analog signal to the digital domain and vise-versa is critical to the miniature RNR's operation. In order to meet the requirement that the proposed system process RF bandwidths of at least 750 MHz, the sampling rates of the data converters will need to exceed 1.5 GS/s. The interface between ultra-high speed data converters and memory is often a significant challenge. For this reason, data converters with buffered or demultiplexed output should be considered. While the selection of ultra-high speed data converters is ever increasing, a good example of an ADC that meets the aforementioned requirements is the MAX109 from Maxim Integrated Products. The MAX109 can achieve sampling rates of 2.2 GS/s and includes a demultiplexer that directs the 8-bit samples to four different output registers. The MAX109's output configuration allows four consecutive samples (32-bits) to be read at one-forth the sampling clock [19].

3.3.3 Clock Generation.

With two ultra-fast data converters and one or more FPGAs, clock generation and distribution is of the utmost importance. The RNR's clock signal needs to be fast enough to drive the ADC and digital-to-analog converter (DAC), and it must have very little phase distortion (low-jitter).

Conventional crystal oscillators usually generate clock frequencies that are well below what is needed for the RNR. As a result, the combination of a crystal oscillator, phase-locked loop (PLL), and a voltage-controlled oscillator (VCO) is often used to meet the clocking requirements of an ultra-high speed data converter. The VCO uses the signal from the crystal oscillator to produce the output clock signal. To keep the VCO's output locked at the desired frequency, the VCO output is fed back to a PLL and compared to the crystal oscillator's frequency. A simple block diagram of the circuit is shown in Figure 3.4 [20].

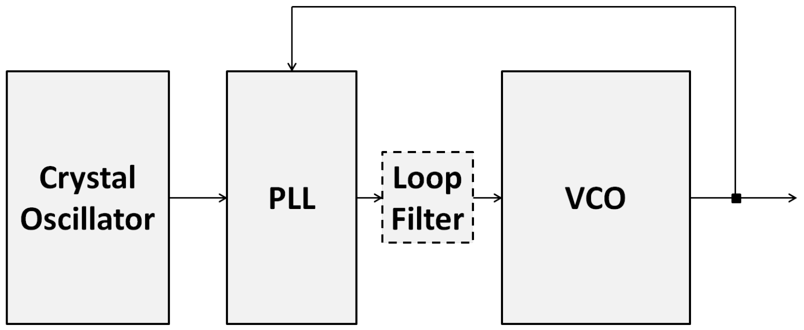


Figure 3.4: Typical Clock for High-Speed Data Converters

To improve the stability of the VCO a loop filter is often used to low-pass filter the signal from the PLL's charge pump. The design of the loop filter is often a balancing act between how quickly the VCO can respond to changes in the PLL signal and stability. To aid in the design process, many manufacturers have free software tools or look-up tables that help identify the necessary filter components to achieved the desired performance.

Once a clock circuit has been designed that can generate the appropriate frequency, special consideration should be given to clock jitter. An irregularity in the clock's period translates to an uncertainty in the quantization of the received signal. An illustration of this phenomena is provided in Figure 3.5. Rapidly changing signals, such as the one used by the RNR, accentuates this uncertainty. Hence, the lower the jitter the better the ADC's SNR. These distortions of the clock's phase and period are the result of internal noise sources: thermal noise, phase noise and spurious noise. A good description of each of these noise sources and their effect on clock jitter is given in Application Note 800 from Maxim Integrated Products [20].

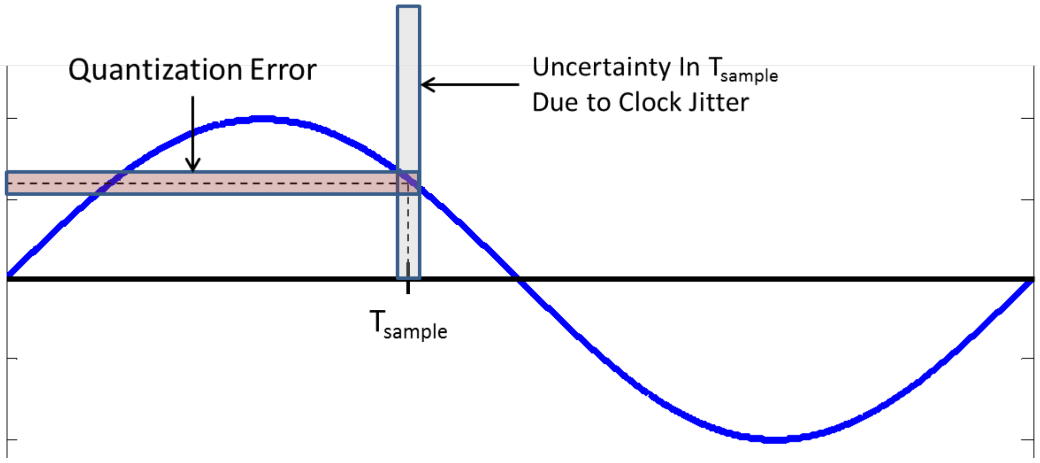


Figure 3.5: Quantization Error Caused by Clock Jitter

3.4 Algorithm Development

The selection of an FPGA as the primary signal processing device has a profound impact on the correlation algorithm that will be used to execute the radar's receive function. To maximize the performance, the design will need to incorporate the FPGA's ability to execute operations in parallel. In other words, the algorithm will need to be highly pipelined. Finally, the correlation routine should efficiently utilize the FPGA's limited resources.

It was shown in Chapter 2 that the cross-correlation of two signals, $x[n]$ and $y[n]$, could be calculated by taking the IDFT of their cross-power spectral density. This method is well suited for FPGA implementation because it is built around the inherent efficiencies of the DFT. For that reason, a hardware realization of the R2^2SDF FFT will be presented below. The FFT module will serve as a cornerstone upon which the rest of the correlation algorithm will be built.

3.4.1 Radix-2^2 Architecture.

A block diagram of the R2^2SDF is presented in Figure 3.6 [31]. An N-point R2^2SDF processor usually contains $\log_4(N)$ stages. Each of the stages contain two

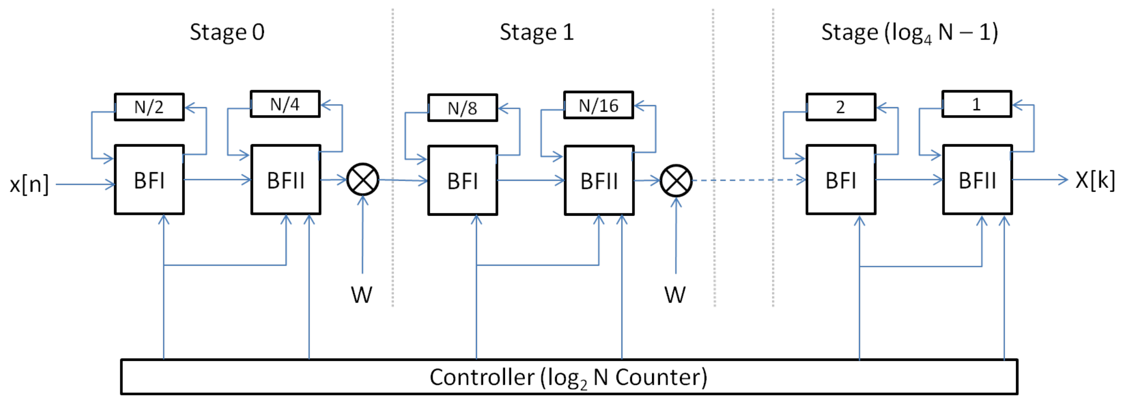


Figure 3.6: R2^2SDF Block Diagram

hardware modules that are designed to compute a two-point DFT and will hereafter be referred to as BFI and BFII respectively (BF is a shortened version of butterfly, a word commonly used to describe the signal flow graph of a two-point DFT). Each of the butterfly units are connected to a series of memory registers that are used to delay the feedback signal. The output of BFII flows through a complex multiplier where it is multiplied by the complex roots of unity (i.e. "twiddle-factors") described in Equation (2.26). The twiddle factors are stored in the FPGA's block random-access memory (RAM). Finally, a $\log_2(N)$-bit counter is used as a control unit for the R2^2SDF processor [14]. In cases where N is not a power of four but is a power of two, the final stage will only contain a BFI module.

A detailed view of the BFI architecture is shown in Figure 3.7 [31]. The control signal, C_1, oscillates between 0 and 1 every $N/2^{stage+1}$ clock cycles. Initially, C_1 is in state 0 and the multiplexers direct the input signal to the delay buffers. When C_1 changes to state 1, a two-point DFT is computed between the input signal and the delayed signal. The real and imaginary outputs of the BFI module are fed to the next component, normally a BFII module.

The schematic of the BFII module is shown in Figure 3.8 [31]. The structure of the BFII is more complex than the BFI as it is responsible for computing the trivial

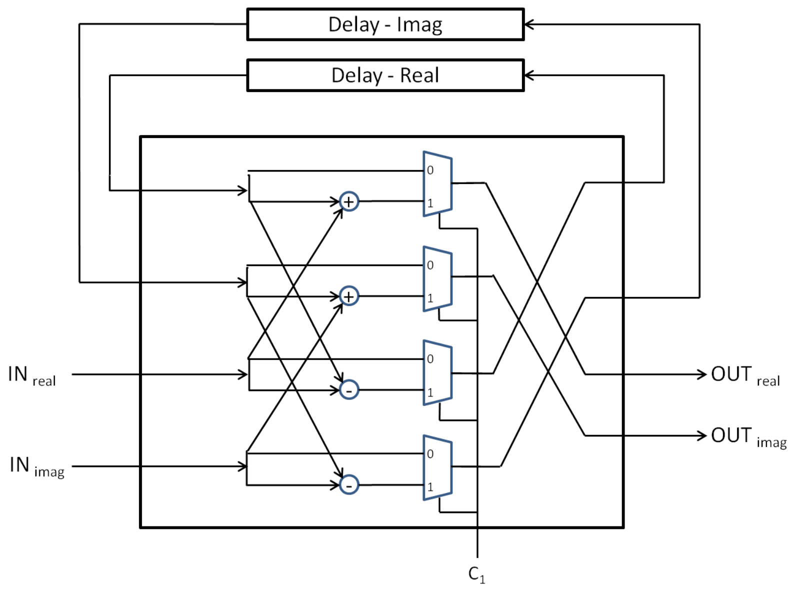


Figure 3.7: BFI Architecture

multiplications by $-j$ prescribed by Equation (2.27). Multiplying a complex number, $A + jB$, by $-j$ results in:

$$-j(A + jB) = B - jA. \tag{3.2}$$

In other words, IN_{real} and IN_{imag} would be swapped and the sign of the real input would be inverted. There are two control signals, C_1 and C_2 used to route the signals through the BFII. The signal C_2 is used direct the output of the MUX_{im} module to the delay buffers or through the two-point DFT butterflies. C_2 changes state every $N/2^{stage+2}$ clock cycles (i.e. twice the rate of C_1). The control signal C_1 is the same input that is used to control the BFI, and is combined with C_2 to control the MUX_{im} and sign inversion modules. A detailed view of the sign inversion module is shown in Figure 3.9 [31].

The next component in the R2^{2}SDF architecture that is worthy of discussion is the complex multiplier. Traditionally the multiplication of two complex numbers is computed as follows:

$$(A + jB)(C + jD) = AC - BD + j(AD + BC). \tag{3.3}$$

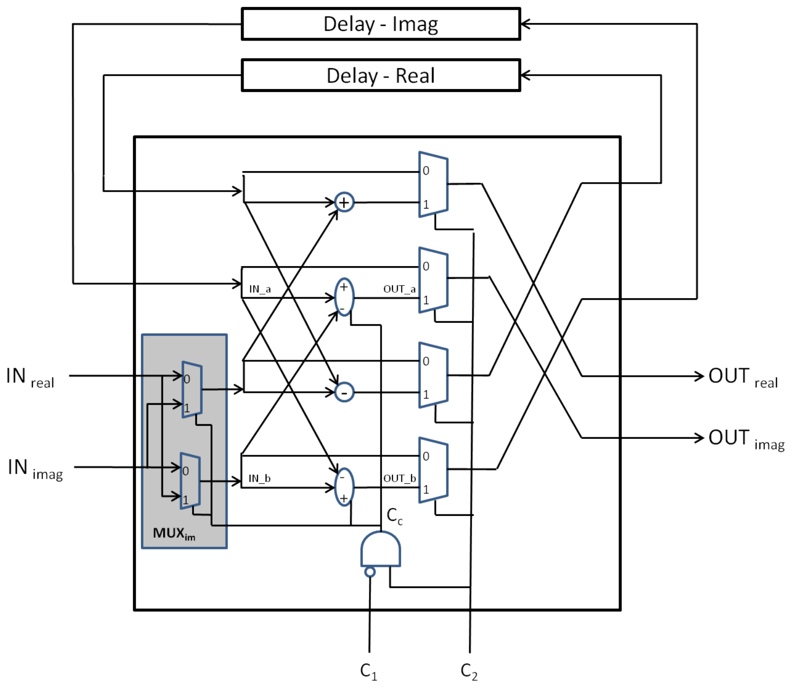


Figure 3.8: BFII Architecture

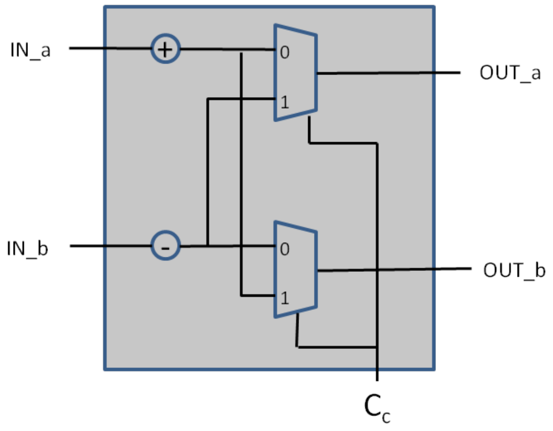


Figure 3.9: BFII Sign Inverter

The end result requires four real multipliers and two real adders. Since FPGA multipliers are often a scarce resource, a secondary approach is often taken to minimize the number of multiplications required for the complex product. If (3.3) is rearranged as follows:

$$(A + jB)(C + jD) = (C(A - B) + B(C - D)) + j(D(A + B) + B(C - D)), \qquad (3.4)$$

the number of additions required is increased to five, but the number of multiplications drops to three. Equation (3.4) results in a computational latency of six clock cycles. A schematic of the pipelined complex multiplier can be seen in Figure 3.10 [33].

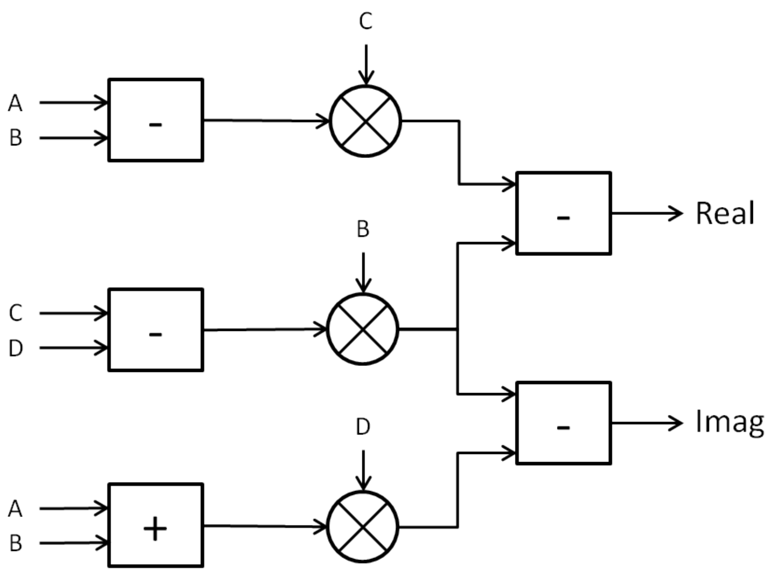


Figure 3.10: Pipelined Complex Multiplier

Another critical component in any FFT algorithm is the twiddle-factor generator. Many different techniques have been proposed throughout existing literature. These include, but are not limited to, coordinate rotation digital computer (CORDIC) algorithms, polynomial based approaches, ROM-based lookup tables, and recursive function generators. CORDIC algorithms are often used in smaller FPGAs to calculate trigonometric functions using a combination of adders, bitshift operations, and lookup tables. While

CORDIC algorithms are efficient, they introduce unnecessary delays in larger, more capable devices like the Xilinx Virtex-5. The polynomial and recursive function approaches use polynomials as a piecewise approximation to complex functions. These algorithms use more resources than the CORDIC-based approach and grow in complexity as precision is increased. Finally, ROM-based lookup tables can be used to store pre-calculated values of the desired function. This approach eliminates the computational latencies in the other algorithms, but can consume copious amounts of memory for larger transforms. The ideal choice depends heavily on the design parameters of the FFT processor. For example, a ROM-based lookup table for an 8192-point transform may consume too much of the FPGA's memory, and CORDIC algorithms may be too slow for high-throughput processors [4].

While the authors of [4] only recommend the ROM-based approach for transform sizes of N = 512 or less, the proposed FFT processor will utilize a lookup table for simplicity. The twiddle-factors are precomputed in MATLAB, converted to fixed-point precision, and then loaded into the FPGA's read-only memory (ROM) upon implementation. The twiddle-factors are generated according to the following algorithm:

$$W_i = \{u_x\}; \qquad x = 0, 1, 2, \ldots, N/2^{2i}, \tag{3.5}$$

$$u_x = e^{-j2\pi v/N}, \tag{3.6}$$

$$v = \begin{cases} 0, & 0 \leq x < a \\ 2^{2i+1}(x-a), & a \leq x < 2a \\ 2^{2i}(x-2a), & 2a \leq x < 3a \\ 3 \cdot 2^{2i}(x-3a), & 3a \leq x < 4a \end{cases} \tag{3.7}$$

$$a = \frac{N}{2^{2(1+i)}}, \tag{3.8}$$

where i corresponds to the current stage and ranges from zero to $log_4(N) - 2$ [31]. If the algorithm is computing the inverse fast Fourier transform (IFFT), the twiddle-factors generated by the above routine would need to be conjugated before being stored to the FPGA.

3.4.2 Bit-Order.

The R2^2SDF FFT processor is an example of a DIF routine. As a result the input signal, $x[n]$, will enter the R2^2SDF algorithm in normal order. Once the transform has been completed, the algorithm's output, $X[k]$, exits the routine in bit-reversed order. Therefore, there are two options that will impact the components that follow the FFT block in the correlation routine: bit-reverse the matched filter and design an entirely new architecture to compute the IFFT or bit-reverse $X[k]$ so that the same R2^2SDF architecture can be used to compute the FFT and IFFT. The first option would reduce the overall latency of the correlation algorithm. However, the challenge of designing a piplined DIT algorithm that operates with the same efficiency as the R2^2SDF is not a trivial task. For that reason, the correlation algorithm presented in this chapter bit-reverses $X[k]$ so the R2^2SDF can be reused.

The hardware module that will be used to execute the bit reversal was introduced Garrido, et al. in [11]. If $N = 2^n$, then a positional vector (i.e. index), P can be defined as

$$P = \sum_{i=0}^{n-1} x_i 2^i. \tag{3.9}$$

Consider an $X[k]$ that has an initial index of $P_0 = u_{n-1}, u_{n-2}, \ldots, u_0$. An elementary bit exchange of two dimensions x_j and x_k, where $j > k$, consists of moving each sample in position P_0 to the new index P_1 where:

$$P_0 = u_{n-1}, u_{n-2}, \ldots, u_{j+1}, u_j, u_{j-1}, \ldots, u_{k+1}, u_k, u_{k-1}, \ldots, u_0 \tag{3.10}$$

$$P_1 = u_{n-1}, u_{n-2}, \ldots, u_{j+1}, u_k, u_{j-1}, \ldots, u_{k+1}, u_j, u_{k-1}, \ldots, u_0. \tag{3.11}$$

In other words, the $N/2$ samples that have an indices $x_j \neq x_k$ will exchange places while the rest are unaffected [11].

In the case of the R2^{2}SDF architecture the samples arrive in a serial manner. Therefore, the position of the sample is equal to the order of arrival, and the number of the clock cycles between x_j and x_k is equal to

$$\Delta t = 2^j - 2^k. \tag{3.12}$$

Exchanging the samples in indices $x_j \neq x_k$ requires delaying some samples while advancing others. This can be accomplished in hardware using two muxes and $L = \Delta t$ registers. Figure 3.11 depicts the basic circuit necessary for exchanging serial dimensions [11]. The

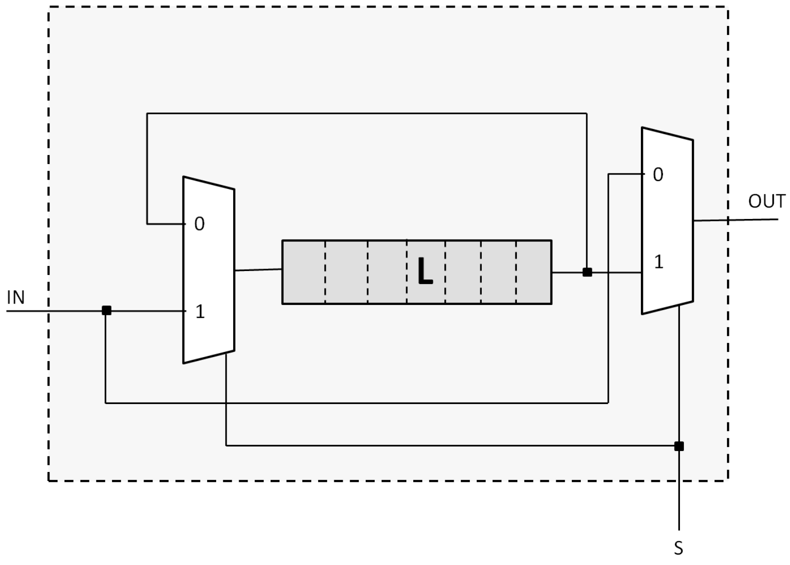


Figure 3.11: Basic Circuit To Exchange Dimensions of Serial Data

control signal, S, is driven by the following logic:

$$S = \overline{x_j} \mathrm{OR} x_k. \tag{3.13}$$

Two samples, $S1$ and $S2$, are exchanged every time $x_j = 1$ and $x_k = 0$.

Now that the basic circuit for dimensional exchange has been identified, a framework for bit-reversal can be developed. Let σ represent the following permutation:

$$\sigma(x_{n-1}, x_{n-2}, \ldots, x_0) = x_0, x_1, \ldots, x_{n-1}. \tag{3.14}$$

Equation (3.14) can be seen as a composition of elementary bit exchanges. Therefore, it can be represented as

$$\sigma_i = \begin{cases} x_i \leftrightarrow x_{n-1-i}\,;\ i \in [0, n/2-1] & \forall \text{ Even } \{log_2(N)\} \\ x_i \leftrightarrow x_{n-1-i}\,;\ i \in [0, (n-3)/2] & \forall \text{ Odd } \{log_2(N)\} \end{cases} [11]. \tag{3.15}$$

The R2^{2}SDF algorithm introduced in this chapter outputs $N = 1024$ samples in bit-reversed order. Hence, to compute

$$\sigma(x_0, x_1, \ldots, x_9) = x_9, x_8, \ldots, x_0 \tag{3.16}$$

a total of 5 exchange circuits are needed. The number of delay registers needed in each σ_i is defined by:

$$D(\sigma_i) = 2^{(n-1-i)} - 2^i \ [11]. \tag{3.17}$$

A complete schematic of the circuit is shown in Figure 3.12.

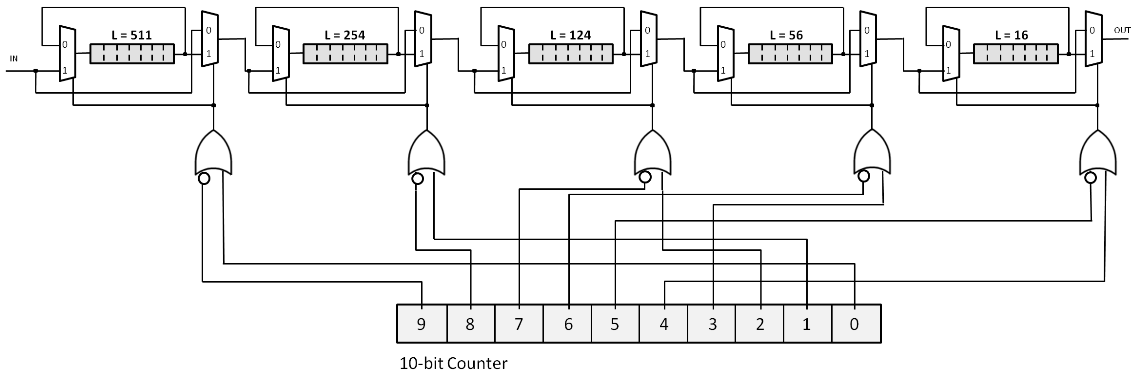


Figure 3.12: 1024-Point Bit-Reverse Circuit

3.4.3 FPGA Correlation.

Now that the architectures of the FFT, complex multiplier and bit-reverser have been established, the FPGA correlation algorithm can be constructed. A block diagram of the proposed algorithm can be seen in Figure 3.13. The real and imaginary components of the

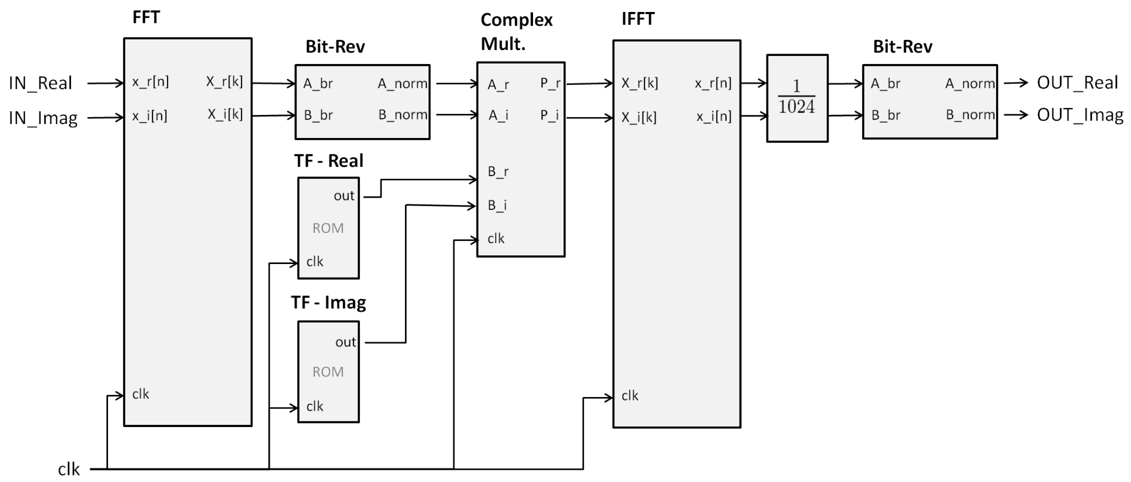


Figure 3.13: Proposed FPGA Correlation Algorithm

input signal are fed into the first R2^{2}SDF FFT. The output of the FFT is then sent through a bit-reversal module to convert the data back into normal serial order. Next, the data stream enters the complex multiplier where it is multiplied by corresponding filter coefficients that have been predetermined and loaded in FPGA ROM. The result is then sent through a second R2^{2}SDF module that has been configured to compute the IFFT. Finally, the data is scaled according to Equation (2.16) and then rearranged back into normal order.

The proposed correlation algorithm assumes that the input is synchronized with the template such that when the signal of interest is received

$$\text{template}[0{:}1023] = (\text{input}[0{:}1023] - \text{U}[0{:}1023])^{*} \tag{3.18}$$

where U is the channel noise. Additionally, an FPGA implementation of the proposed algorithm has to account for the latencies of each component so that all of the mathematical operations are appropriately aligned.

3.5 Design Tools

In order to develop a simulation and hardware implementation of the proposed correlation algorithm, a few design tools were needed. Below is a brief description of those that were used during this research effort.

3.5.1 Simulink.

Simulink is a software tool developed by MathWorks that uses a graphical design environment to model, simulate and analyze dynamic systems across many different domains. Simulink is closely coupled with MATLAB. This allows Simulink models to be run from MATLAB scripts and for the data generated within a Simulink model to be exported to MATLAB for further analysis. The solvers within Simulink are capable of handling continuous-time and discrete-time models.

3.5.2 Xilinx ISE.

The ISE Design Suite from Xilinx is a comprehensive integrated development environment (IDE) that facilitates FPGA design from start to finish. There are several specialized versions of the ISE Design Suite that have been tailored to specific design paradigms (i.e. embedded systems, DSP, etc.) and include support for a wide range of Xilinx hardware. In addition to licensed versions above, the ISE Webpack is a free version of the software available for download on the company's website. The licensed versions of the software and the ISE Webpack provide developers with a library of pre-developed FPGA modules known as "intellectual property (IP) cores". A thorough description of the development environment and related software modules can be found in [42, 41].

The ISE Webpack, version 14.4 was chosen for this development effort because it included support for AFIT's Virtex-5 (XC5VLX50T-FFG1136C -1). Furthermore, the standard set of IP cores that is included under the Webpack license was sufficient for the proposed correlation algorithm.

3.5.3 ML555: Virtex-5 Development Board.

To facilitate the design of the correlation algorithm on an FPGA, Xilinx's Virtex-5 ML555 development kit was employed. The ML555's primary purpose is the design of FPGA algorithms for parallel peripheral component interface (PCI) and serial PCI Express communication. However, the development board presents developers with several options for IO. The ML555 development board is pictured in Figure 3.14. To operate the ML555 it must be properly configured and then inserted into a PCI or PCI Express add-on slot. The board includes three push button switches and three light emitting diodes that allow users to interact directly with their algorithm. The ML555 has three global clock sources, a differential subminiature version A (SMA) clock input, and two programmable clock sources [43].

Bitstreams can be programmed directly to the FPGA via the Xilinx Platform Cable universal serial bus (USB). Conversely, the bitstream files can be converted to a format compatible with the ML555's Platform Flash via the Xilinx software module iMPACT. The development board includes a complex programmable logic device (CPLD) that can be configured as a bootloader for FPGA programs stored in the two Platform Flash units. The two Platform Flash units can support a total of four different bitstreams [43].

3.6 System Characterization

Now that a design has been proposed for a correlation algorithm and the required tools have been identified, the next step is to characterize the system against the key system attributes for the miniature RNR's receiver.

3.6.1 Modeling and Simulation.

A model of the proposed algorithm will be designed in Simulink. A simulation of the correlation algorithm will be used to determine if the proposed design delivers a valid

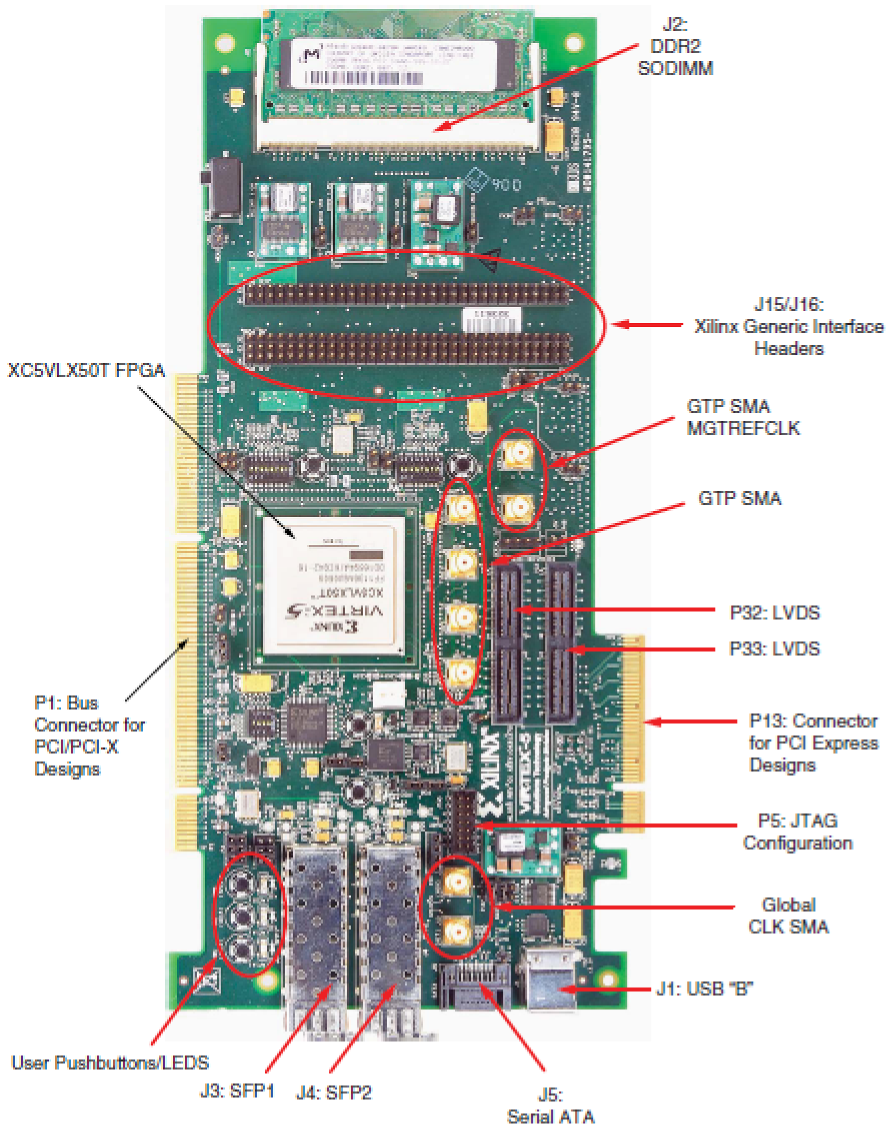


Figure 3.14: Annotated ML555 Board [43]

result. If problems are encountered, the necessary changes will be made to the model before proceeding to FPGA design.

The one drawback to this approach is that AFIT's license does not currently cover MATLAB's fixed-point toolbox. Hence, all simulations will be executed at double precision. Design features that handle overflow control, loss of precision, and rounding will have to be addressed during hardware description language (HDL) coding.

3.6.2 Power Assessment.

The ML555 provides current sensing circuits for monitoring the current required to power the FPGA and select peripherals [43]. These circuits will be used in conjunction with a volt-ohm meter to characterize the power required by the system when idle and when executing the correlation algorithm. The difference should provide some insight to the power required to run the routine.

In addition to the direct measurement described above, the Xilinx ISE is able to estimate the power required to run a given design. The Xilinx program, XPower Analyzer, is included with the ISE Webpack installation. The XPower Analyzer provides a detailed report of the power consumed by a given design once it has been implemented and a successful place and route has been accomplished. For an accurate measurement, the program requires a well defined user constraint file to get an idea of net toggle rates. The XPower Analyzer report can be used to highlight areas of the design (specific hardware or code modules) that are consuming the most power and have the greatest potential for energy savings. The results provided by this program should provide a good complement the ML555's current sensing circuits.

3.6.3 Performance Assessment.

To determine if the proposed correlation algorithm is a good candidate for future RNR miniaturization efforts the system's performance will need to be assessed. The current

AFIT RNR will be used as a performance baseline and comparisons will be made to determine the suitability of an FPGA-based receiver.

The first attribute that is of interest is the computational speed of the proposed algorithm. For the proposed design to be considered a success, it must achieve a computational rate at least equal to that of the current AFIT RNR.

The ML555 development board does not provide a convenient way to send a timing strobe to a logic analyzer or oscilloscope. For this reason, an alternative approach will be taken to determine the time required to complete a correlation. In the top-level module of the FPGA correlation algorithm, a memory register will be added that begins counting clock cycles when the start signal is asserted and stops after the result is stored in the FPGA's memory. The count value is then sent to a computer where it is multiplied by the clock period to determine the overall time elapsed.

Another important aspect of the algorithm's performance is accuracy. To assess the ability of the proposed algorithm to provide the correct result, an input signal will be generated in MATLAB and stored in the FPGA's ROM. An arbitrary 1024-point sample will be taken from the input signal to generate coefficients of the matched filter. When the start signal is asserted the input signal will flow through the correlation algorithm and the result will be stored in RAM. Upon completion of the transform, the result will be sent to a computer and plotted in MATLAB. Finally, the result of this test will be compared to a similar correlation routine in MATLAB.

3.7 Conclusion

The selection of an FPGA centric design for the miniature RNR led to the development of a correlation algorithm. Once a model of the proposed routine is proven in Simulink, an FPGA implementation will be used to assess the performance of the proposed algorithm. The result of these evaluations will be discussed in the next chapter.

IV. Results

4.1 Chapter Overview

THE R2^{2}SDF correlation routine that was introduced in the previous chapter was simulated and then implemented in actual hardware. Each of the assessments described in Chapter 3 were performed and the results are presented in the sections below. Comparisons are made between the computer-based approach used in the prior RNR architecture and the simulated/measured results of the proposed system. The chapter begins with a description of the R2^{2}SDF -based correlation model, followed by a description of the FPGA implementation of the routine and concludes with the presentation of the measured results.

4.2 Modeling and Simulation

Before engaging in hardware implementation, a the R2^{2}SDF correlation routine was modeled in Simulink. Simulink was chosen as the modeling and simulation vehicle over HDL simulators like Xilinx's iSim because it provided an interface to MATLAB for plotting and analysis, changes/corrections were much quicker in Simulink, and familiarity alleviated the steep learning curve associated with the HDL simulator. The model served two purposes: it provided valuable insight into the inner workings of the proposed algorithm, and it ensured that it could produce results that closely agreed with theory.

4.2.1 Model Description.

A top-level view of the correlation routine is provided in Figure 4.1. The input signal is generated in MATLAB and loaded into the model via the workspace. The input signal flows through the FFT and bit-reversal subsystems where it is then multiplied by filter coefficients. The filtered result is inverse transformed and then sent through a second bit-reversal subsystem to put the result in normal order. The simulation output is sent to the MATLAB workspace where further analysis can be carried out.

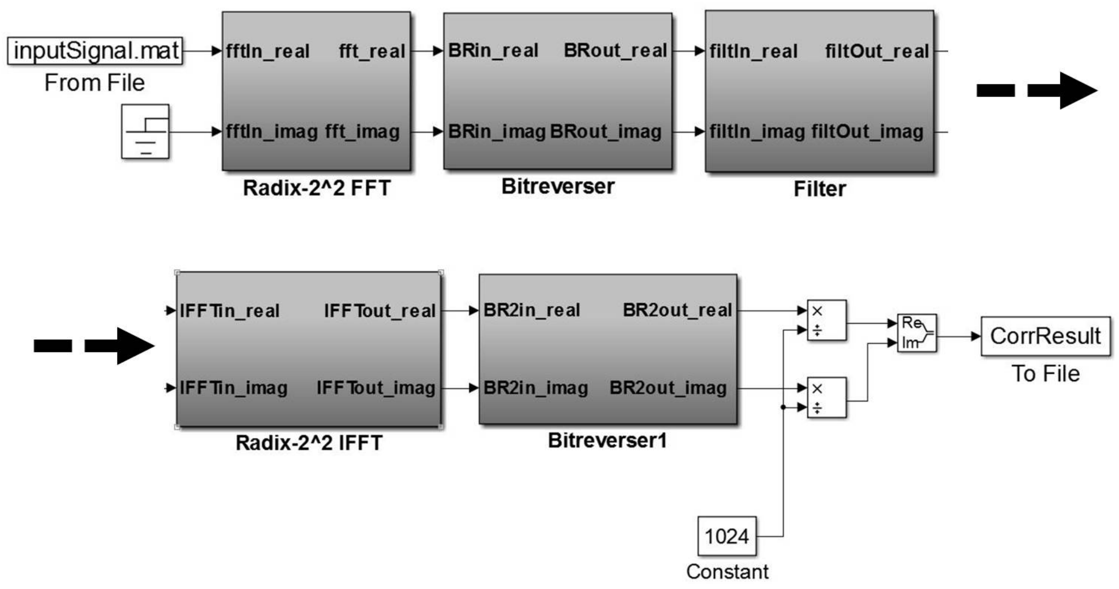
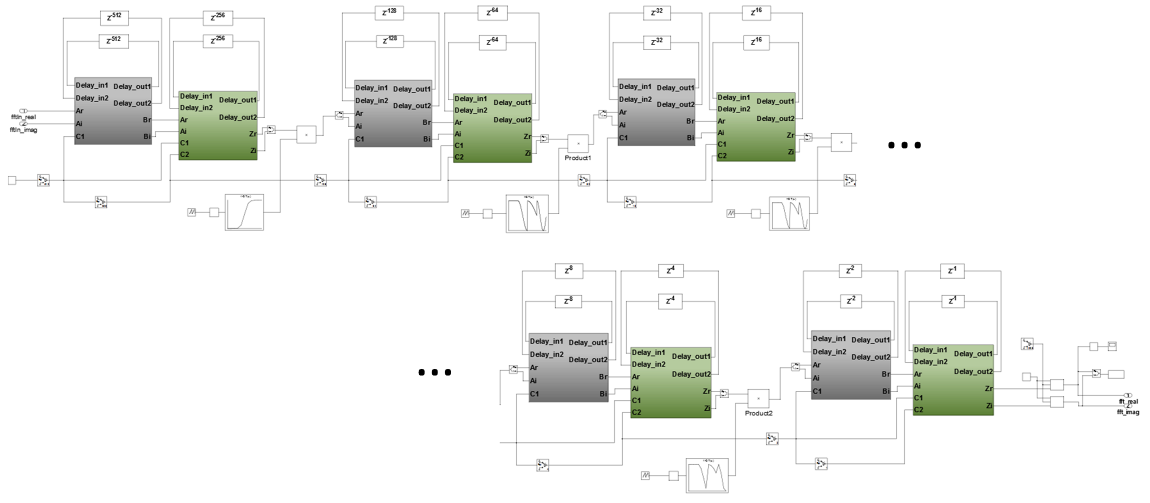


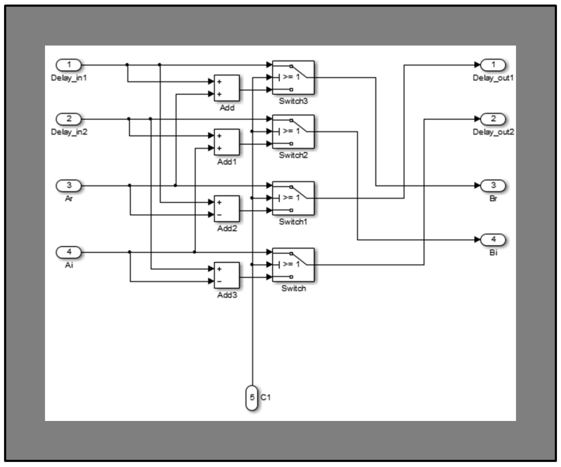
Figure 4.1: Simulink Model of the Correlation Algorithm

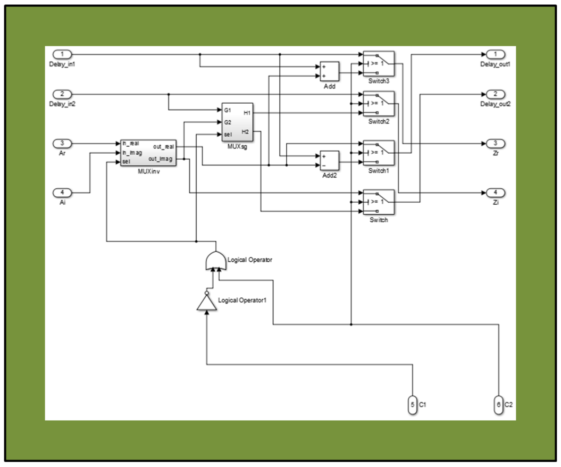
The R2^2SDF FFT subsystem is modeled after the description given in Chapter 3 and can be seen in Figure 4.2. There are a total of five stages, each containing a BFI subsystem, BFII subsystem, and delay units. The output from the first four stages flows through a complex multiplier where it is multiplied by twiddle factors. The twiddle-factors were computed in MATLAB and then stored as MATLAB data files. To successfully execute the FFT, the twiddle-factors have to be loaded to the MATLAB workspace before running the model. The model for the inverse transform has the exact same structure except that the twiddle-factors are the conjugates of those used in the forward transform.

The Simulink design elements, “N-sample enable” and “pulse generator”, have been combined to emulate a 10-bit counter that serves as the control unit for the FFT routine. However, this implementation of a counter results in an inverted output (i.e. 0x0 is



(a)





(b) (c)

Figure 4.2: Simulink Model (a)R2^{2}SDF FFT, (b)BFI Model and (c) BFII Model

represented as 0xF). The inverted control signals and the implementation of Simulink switches result in butterfly units that are slightly different than what was shown in Chapter 3. The logic table for the MUXinv has to be inverted to maintain proper flow through the BFII subsystem. Table 4.1 illustrates the difference between the Chapter 3 MUXinv logic and the Simulink implementation. To achieve the desired result an OR gate is used in place of the AND gate shown in Chapter 3. The Simulink representation of the BFI and BFII can be seen in Figures 4.2b and 4.2c respectively.

Table 4.1: Logic Tables for MUXinv

Chapter 3				Simulink			
C1	C2	Cc	MUXinv Status	C1	C2	Cc	MUXinv Status
1	0	0	Normal	0	1	1	Normal
1	1	0	Normal	0	0	1	Normal
0	0	0	Normal	1	1	1	Normal
0	1	1	Switch	1	0	0	Switch

Following the R2^2SDF FFT in the correlation model is the Simulink model of the bit reversal circuit described in Chapter 3. A top-level view of the bitreverser is shown in Figure 4.3. For the bitreverser to successfully reorder the incoming data, the module's control signals need to be synchronized with the index of the incoming data. Since the latency of the FFT subsystem is exactly 1024 clock cycles, aligning the bitreverser with the incoming signal is quite simple. A ten-bit counter that increments on every clock cycle will return to zero as soon as the first input sample enters the bit reversal module. Hence, the counter is the only control element needed.

Within the bit reversal subsystem, there are two lower-level routines that operate on the real and imaginary data streams respectively. The two submodules are identical and are a Simulink implementation of the circuit described in Section 3.4.2. The constituents of this routine are shown in Figure 4.4. The Simulink implementation of the bit reversal routine is a very close representation of the circuit described in Section 3.4.2.

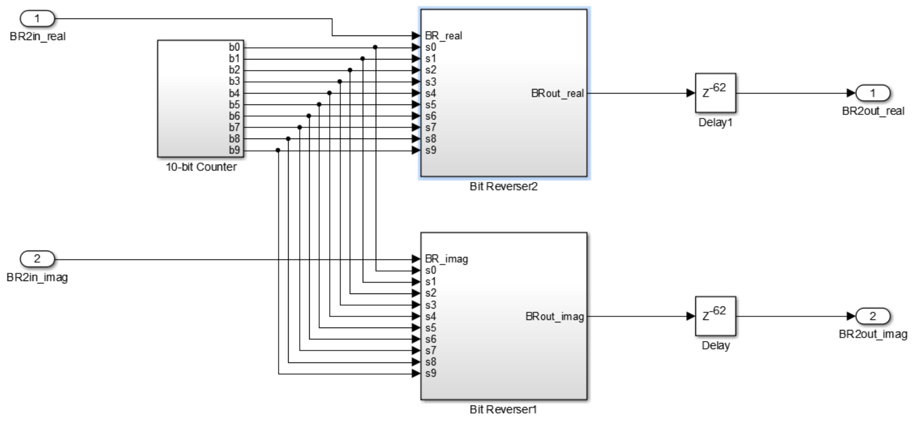


Figure 4.3: Simulink Bitreversal Unit

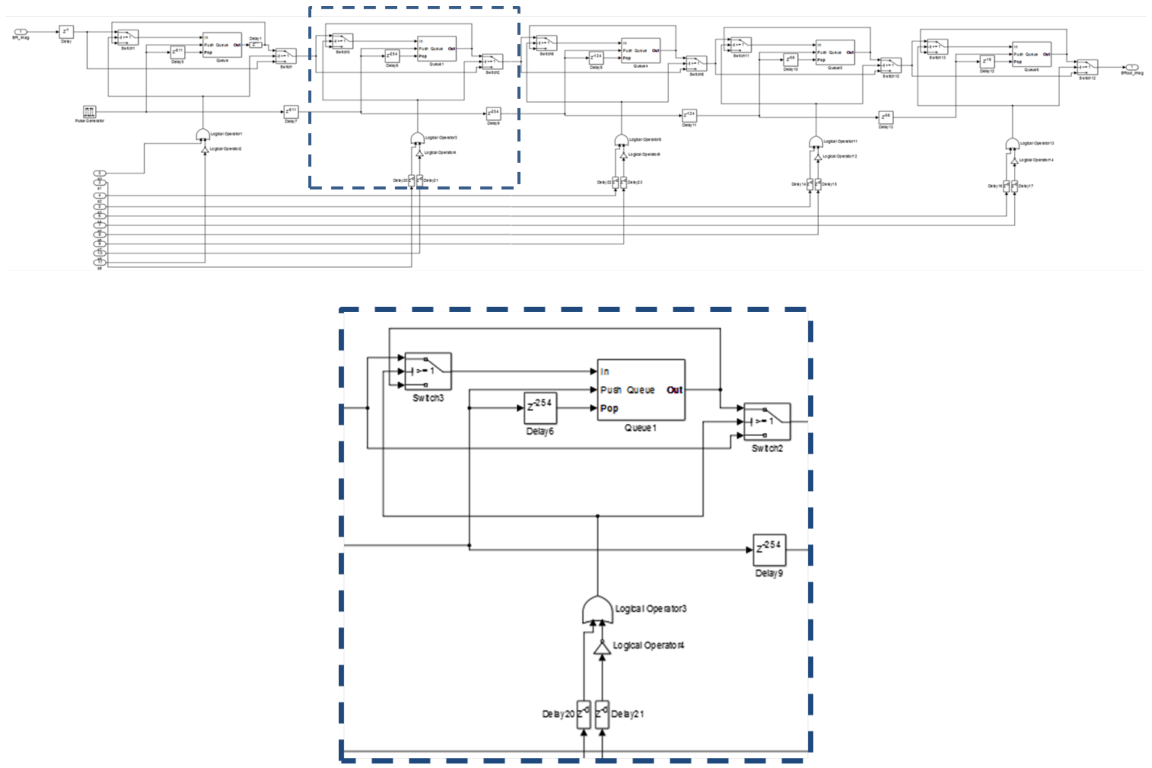


Figure 4.4: Simulink Model of the Bit Reversal Circuit

The latency of the bit reversal routine is 962 clock cycles. Thus, a delay of 62 clock cycles is added to the output in order to increase the overall latency of the bitreverser subsystem to 1024 clock cycles. The added delay allows the downstream subsystems to maintain simple control elements.

The final component in the correlation model is the filter subsystem. The complex filter coefficients are stored in a single lookup table. Because the overall latency of the subsystems preceding the filter is a multiple of 1024, a ten-bit counter can be used to access the data in the lookup table. The real and imaginary components of the input signal are combined into a single complex value and then multiplied by the corresponding filter coefficient. The Simulink model of the filter is shown in Figure 4.5.

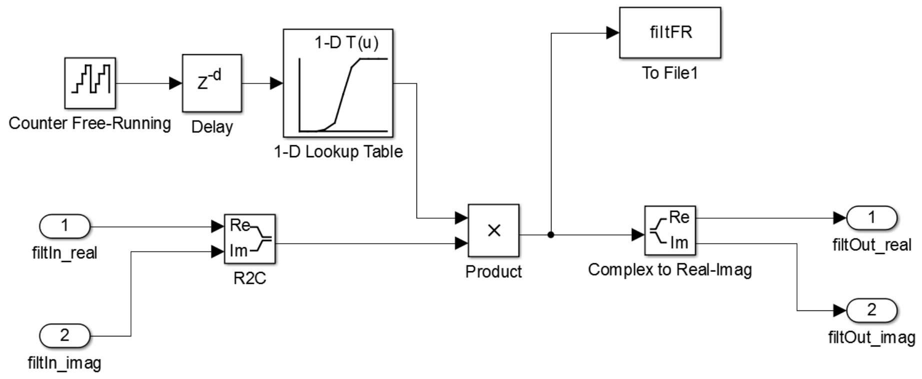


Figure 4.5: Simulink Model of the Filter

4.2.2 Simulation Results.

To demonstrate the correlation model, a random 10240-point input signal was generated using MATLAB's randn function. The input signal was designed to represent ten consecutive captures of an RNR return in a noiseless channel. Each representation of the captured signal is assumed to be 1024 samples long. Two experiments were run on the Simulink model to determine if the R2^{2}SDF FFT and correlation algorithm generated results that agreed with MATLAB.

To verify that the Simulink model of the R2^{2}SDF FFT provides an expected result, the simulation output was plotted along side MATLAB's FFT. The result is shown in Figure 4.6. The smaller plot in the upper right-hand corner of Figure 4.6 shows that the two plots are the same. A mean square error of approximately 6×10^{-29} was calculated, but that could be attributed to machine precision.

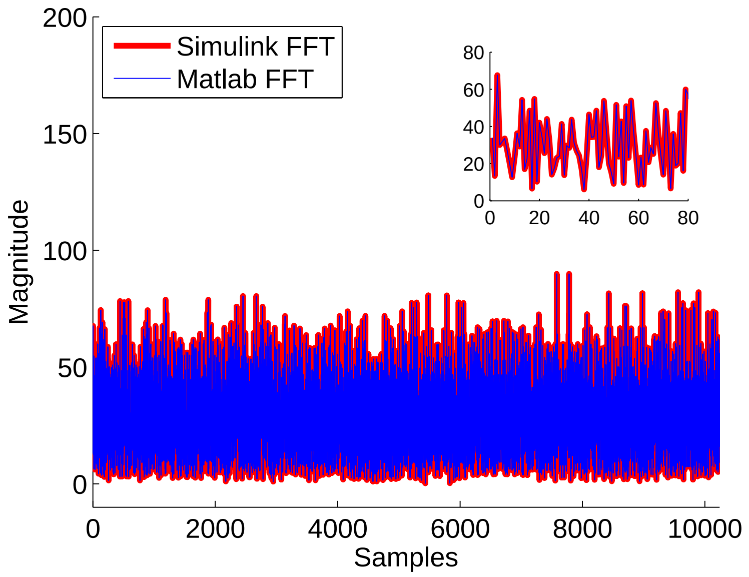
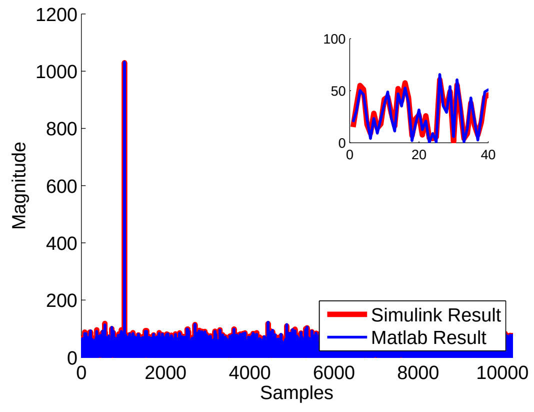
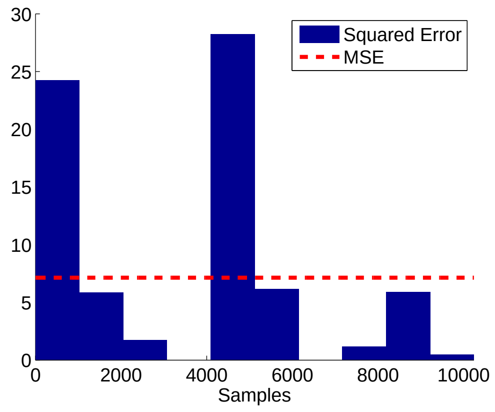


Figure 4.6: Comparison of R2^{2}SDF FFT in Simulink and MATLAB FFT (The plot in the upper right-hand corner is a closeup of the first 80 samples to illustrate the agreement between the two results)

Once the R2^{2}SDF FFT had been shown to produce a valid transform of the input data, the attention shifted to the correlation algorithm. The coefficients for the matched filter were calculated by taking one of the ten captures and conjugating its Fourier transform. The second collection (input(1025:2048)) was arbitrarily chosen for this experiment. Upon completion of the simulation, the result was written to a file and was later plotted against the MATLAB result. Figure 4.7a shows the comparison of the correlation algorithm to a straight forward MATLAB calculation. The two appear to be highly correlated. However, the filter and inverse transform does introduce a small error in the final result. Small differences can be observed in the upper right-hand corner of Figure 4.7a. To explore the error further, a plot of the squared difference between the two is presented in Figure 4.7b. The mean-squared-error was calculated at approximately 7.1 and is shown by the red dotted line in Figure 4.7b.



(a)



(b)

Figure 4.7: Simulation Results (a) Correlation Algorithm vs MATLAB (The plot in the upper right-hand corner is a closeup of the first 40 samples to illustrate the agreement between the two results) (b) Simulation Error

Despite the minor differences in the simulation and MATLAB results, the correlation algorithm has been shown to properly identify the portion of the return signal that corresponded to the filter. The success of the Simulink model provided the justification necessary to pursue an FPGA implementation of the correlation algorithm.

4.3 FPGA Implementation

The ISE Webpack from Xilinx was used to develop an HDL design of the correlation algorithm. The target hardware for FPGA implementation was the ML555's Virtex-5. The HDL code for the correlation algorithm consists of a combination of Verilog, schematic-based coding, and Xilinx IP Cores. Verilog is an HDL programing language that is used to establish a hierarchy of modules and describes the propagation of signals through the design. Verilog is an attractive choice for FPGA coding because it has a syntax that is similar to the C programming language and includes statements that are directly synthesizable. In addition to Verilog coding, the Xilinx ISE includes a graphical method for representing the register-transfer level circuits needed for synthesis. This schematic based design option lets users build modules similar to the way they would in Simulink. Finally, Xilinx provides designers with a library of proprietary black box modules that can be used within a top-level design to carry out common FPGA routines. Because the ML555 includes a Virtex-5, several of these IP Cores were used in the top-level design because it reduced the overall design time for the correlation algorithm and took advantage of the manufacturer's efforts to optimize the design for the target device. A schematic of the top-level module is provided in Figure 4.8.

The primary purpose of the architecture shown in Figure 4.8 is to demonstrate the FPGA-based correlation of an input signal to a single matched filter. The design's similarity to the Simulink model is key to the algorithm's verification as it enables a straight-forward comparison of the two results. To provide a surrogate for an ADC, the input signal is loaded into the FPGA's ROM and streamed through the correlator one sample at time. The same process that was used to generate the input signal and filter coefficients for the Simulink model was also used for the FPGA correlator. However, the 64-bit double precision values produced by MATLAB were converted to signed 16-bit integers. To avoid overflow while maximizing the precision of the input signal, five integer bits and 11 fractional bits were

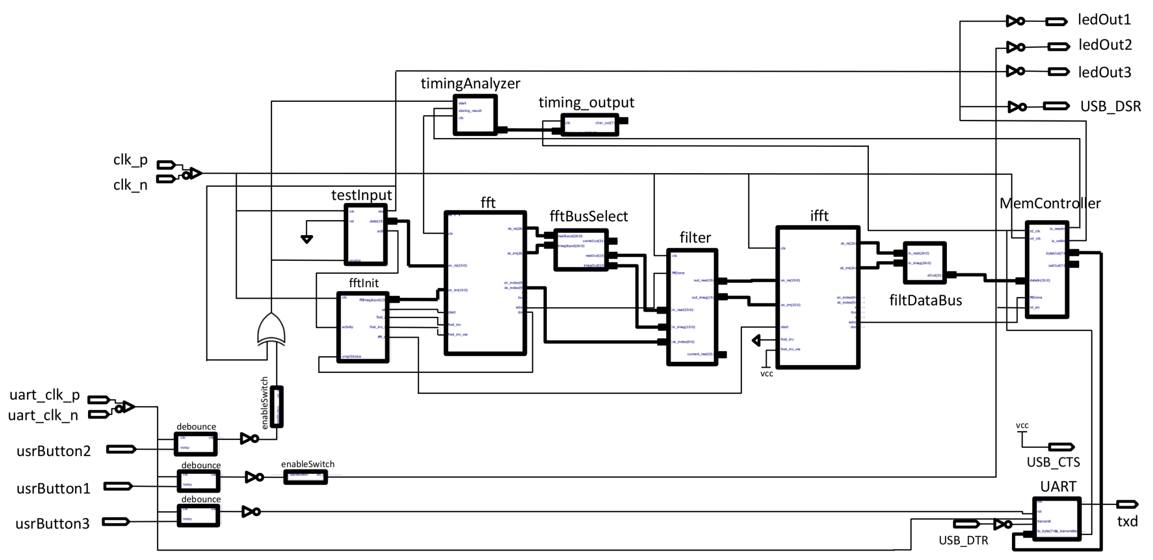


Figure 4.8: FPGA Correlator Schematic

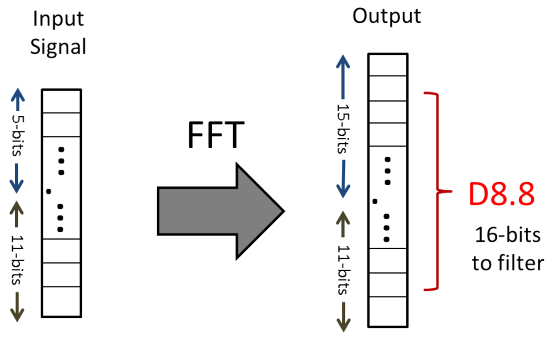
used to represent each value of the fixed-point signal array. A similar approach was taken for the filter coefficients. The larger values in the filter array required eight bits to represent the integer values leaving eight bits for the fractional part.

The ML555's two programmable clock sources are used to generate the differential clock inputs seen in Figure 4.8. The first clock is programmed to generate a 50 MHz clock signal that drives the communications interface. The two differential inputs are tied to pins H19 and H20 on the Virtex-5. The second programmable clock was configured to provide a 125 MHz clock to the correlation algorithm. The leads for this clock are connected to pins J20 and J21.

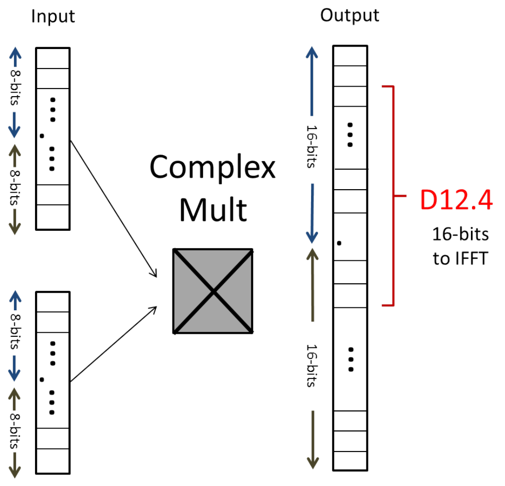
To reduce the design and debugging time required for the FPGA implementation of the correlation algorithm, Xilinx's FFT IP Core was used to compute the routine's forward and inverse transforms. While the literature does not explicitly describe the IP Core as an R2^2SDF FFT, it is a pipelined design with $\log_4(N)$ stages. The benefits of utilizing the IP Core are as follows: it contains a built-in bit reversal routine, there are control outputs that allow the operation to be synchronized with interface modules, and the designer's have ensured that the implementation is optimized for the target FPGA. Follow-on efforts could simply substitute the R2^2SDF routine in the place of the FFT IP Core for FPGAs produced by manufacturers other than Xilinx. IP Cores were also employed for the correlator's memory components and the complex multiplier. While a tailored approach may have reduced resource utilization, the design process is considerably longer.

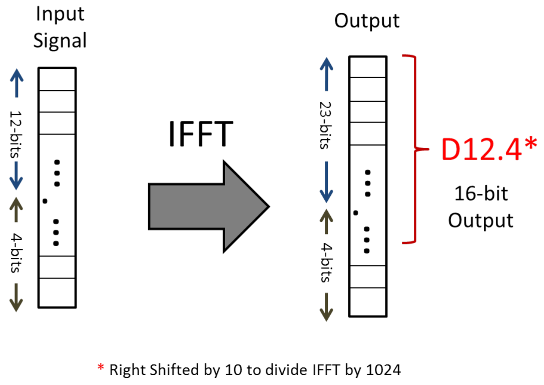
When generating the Xilinx FFT IP Core, an option is provided for scaled or unscaled arithmetic. The scaled option keeps the output in the same format as the input. To avoid overflow, the output from each fixed-point operation is shifted to the right and a rounding scheme is applied to the least-significant bit. However, erroneous results were observed when attempting to calculate the FFT of the input data. Because of the problems observed with the scaled option, the FFT module was configured to provide unscaled output. When configured for unscaled arithmetic, the output from the FFT module experiences a bit-growth from 16-bits to 27-bits in order to avoid overflow. For clarity an illustration is provided in Figure 4.9a. The bit-growth, however, was too conservative and the output was only observed to utilize eight integer bits. Hence, eight of the eleven fractional bits were kept to generate the 16-bit input to the filter module. Similar fixed-point considerations were given to the complex multiplier and IFFT.

A universal asynchronous receiver/transmitter (UART) module is included in the design to conduct serial communications. The UART module serves as an interface between the FPGA and the ML555's UART-to-USB bridge (CP2102). The CP2102 enables a computer to interact with the FPGA via a USB connection. Silicon Laboratories, the manufacturer of the CP2102, provides a device driver that allows the ML555 to appear as a COM port in the computer's device manger. As the correlation routine executes the results are stored inside the FPGA's RAM. An indication is given to the user when the results are ready to be accessed via one of the ML555's light-emitting diodes (LEDs). When



(a)





(b) (c)

Figure 4.9: Fixed-Point Manipulations in the Correlation Algorithm (a) FFT, (b) Complex Multiplier, and (c) IFFT

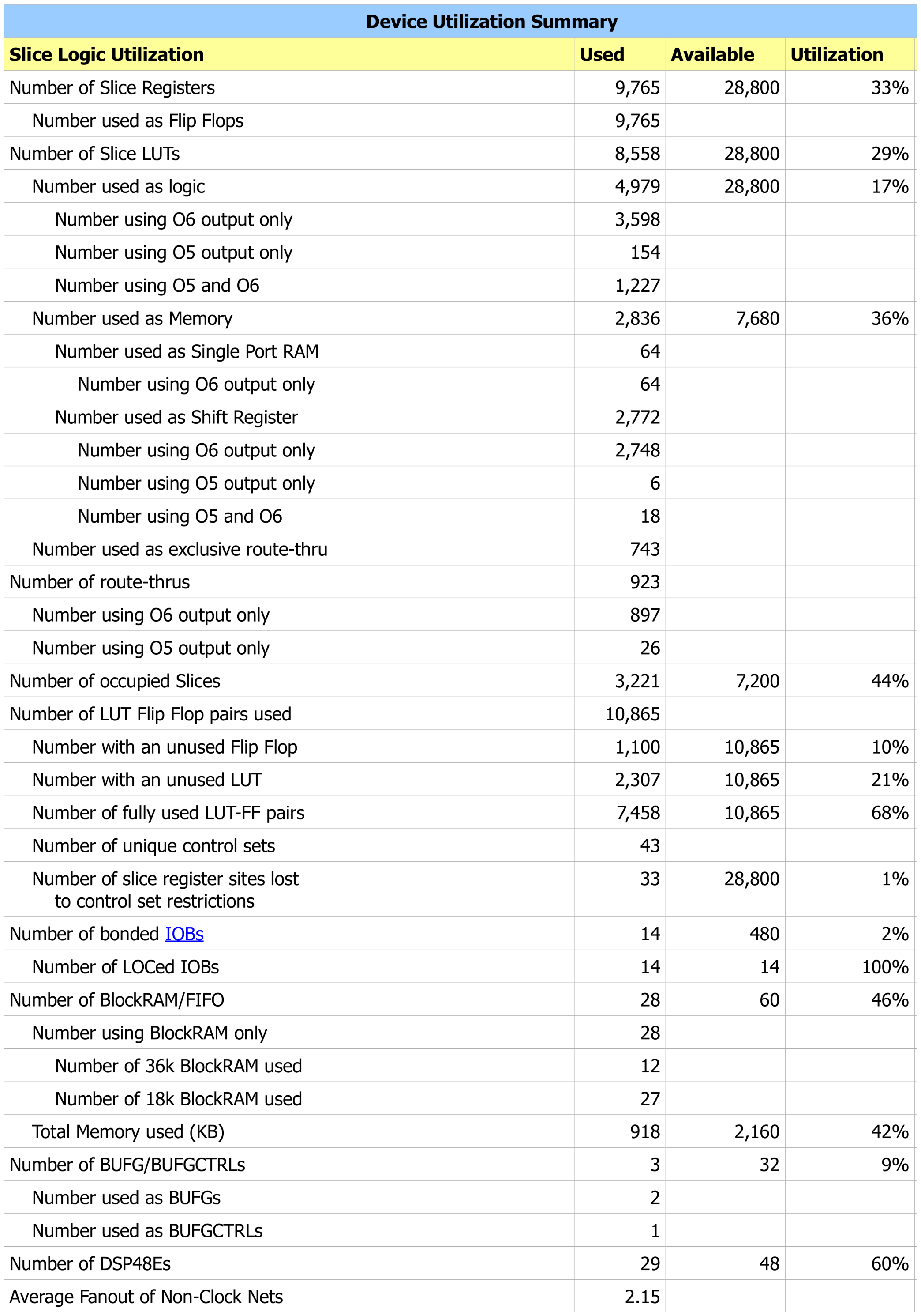
the development board's AF20 push button is pressed, the algorithm's memory controller converts the hex values into its ASCII representation and then transmits the result to the computer one byte at a time. The serial communication is configured for a BAUD rate of 115200 and inserts two stop bits between each byte.

Once the code for the algorithm had been written, the Xilinx ISE was used to synthesize (i.e., compile) and implement the design shown in Figure 4.8. Following the software's place and route routine, a post-implementation report was generated. Figure 4.10 is the resource utilization summary for the FPGA correlation routine. It is important to note that the percentages given in the resource utilization summary includes the additional overhead associated with storing the input signal on the FPGA. While the correlation routine consumes 60-percent of the Virtex-5's DSP specific slices, less than half of the logic slices are occupied. This is an encouraging result as a significant portion of the Virtex-5's resources would be available for ancillary modules like transmitter logic or additional correlation channels.

The post-implementation timing analysis reported zero errors for a system clock of 125 MHz. The delays for each of the signal paths were calculated and the longest was reported to be 6.721 nano-seconds. Hence, the design's final implementation could operate at a maximum frequency of 148 MHz. It is important to note that this number is derived from the propagation delays of the placed and routed design and uses software models of the FPGA's components to estimate the maximum delay. Actual hardware implementation would be required to determine if the proposed design can operate at the maximum frequency.

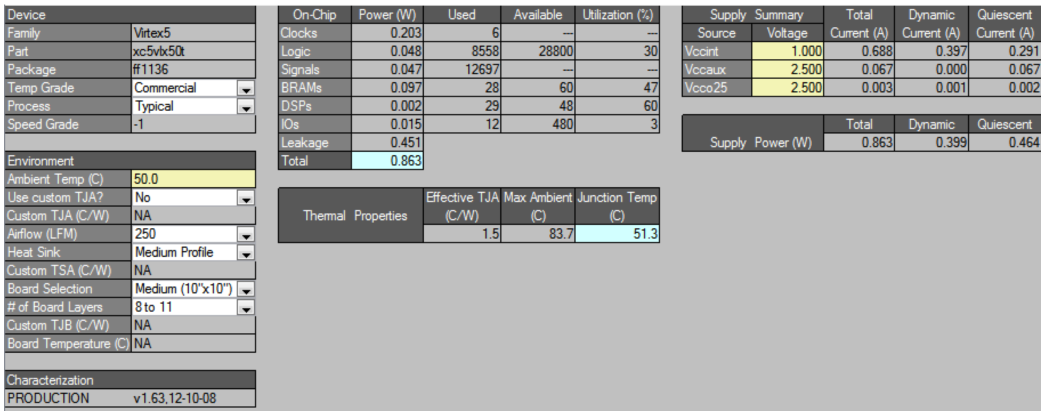
To estimate the power consumed by the correlation routine the Xilinx XPower Analyzer was executed. A summary of the program's analysis is depicted in Figure 4.11. When idle, the design consumes just under half a Watt. However, power consumption rises to 863 mW when the algorithm is executing.

To verify the amount of power consumed by the correlation algorithm, the ML555 development board provides developers access to several current sensing resisters. These measurement pins can be seen in Figure 4.12. A digital multimeter was used to measure the voltage drop across each of the resisters to determine the current. The power consumed at each location can then be calculated by: $P_{\text{consumed}} = I * V$. Table 4.2 summarizes the results.



Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	9,765	28,800	33%
Number used as Flip Flops	9,765		
Number of Slice LUTs	8,558	28,800	29%
Number used as logic	4,979	28,800	17%
Number using O6 output only	3,598		
Number using O5 output only	154		
Number using O5 and O6	1,227		
Number used as Memory	2,836	7,680	36%
Number used as Single Port RAM	64		
Number using O6 output only	64		
Number used as Shift Register	2,772		
Number using O6 output only	2,748		
Number using O5 output only	6		
Number using O5 and O6	18		
Number used as exclusive route-thru	743		
Number of route-thrus	923		
Number using O6 output only	897		
Number using O5 output only	26		
Number of occupied Slices	3,221	7,200	44%
Number of LUT Flip Flop pairs used	10,865		
Number with an unused Flip Flop	1,100	10,865	10%
Number with an unused LUT	2,307	10,865	21%
Number of fully used LUT-FF pairs	7,458	10,865	68%
Number of unique control sets	43		
Number of slice register sites lost to control set restrictions	33	28,800	1%
Number of bonded IOBs	14	480	2%
Number of LOCed IOBs	14	14	100%
Number of BlockRAM/FIFO	28	60	46%
Number using BlockRAM only	28		
Number of 36k BlockRAM used	12		
Number of 18k BlockRAM used	27		
Total Memory used (KB)	918	2,160	42%
Number of BUFG/BUFGCTRLs	3	32	9%
Number used as BUFGs	2		
Number used as BUFGCTRLs	1		
Number of DSP48Es	29	48	60%
Average Fanout of Non-Clock Nets	2.15		

Figure 4.10: Resource Utilization Summary for the FPGA Correlator



Device	
Family	Virtex5
Part	xc5vlx50t
Package	ff1136
Temp Grade	Commercial
Process	Typical
Speed Grade	-1

Environment	
Ambient Temp (C)	50.0
Use custom TJA?	No
Custom TJA (C/W)	NA
Airflow (LFM)	250
Heat Sink	Medium Profile
Custom TSA (C/W)	NA
Board Selection	Medium (10"x10")
# of Board Layers	8 to 11
Custom TJB (C/W)	NA
Board Temperature (C)	NA

Characterization	
PRODUCTION	v1.63,12-10-08

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.203	6	---	---
Logic	0.048	8558	28800	30
Signals	0.047	12697	---	---
BRAMs	0.097	28	60	47
DSPs	0.002	29	48	60
IOs	0.015	12	480	3
Leakage	0.451			
Total	0.863			

Thermal Properties	Effective TJA (C/W)	Max Ambient (C)	Junction Temp (C)
	1.5	83.7	51.3

Supply Summary Source	Voltage	Total Current (A)	Dynamic Current (A)	Quiescent Current (A)
Vccint	1.000	0.688	0.397	0.291
Vccaux	2.500	0.067	0.000	0.067
Vcco25	2.500	0.003	0.001	0.002

	Total	Dynamic	Quiescent
Supply Power (W)	0.863	0.399	0.464

Figure 4.11: XPower Analyzer Report for the Correlation Algorithm

4.4 Accuracy of the Correlation Algorithm

The input signal and template were generated to have the same characteristics as the Simulink model. The input signal consisted of ten 1024-point collections for a total of 10240 data points. Furthermore, there was no noise added to the input signal providing the algorithm with the ideal conditions for correlation. The coefficients for the matched filter were generated from the second collection (inputSignal(1025:2048)). As mentioned in Section 4.3, the input signal and filter coefficients consisted of 16-bit signed integers. The input was of the format Q4.11 (one sign, four integer, and eleven fractional bits) and the filter coefficients were formatted as Q7.8.

To execute the program, the ML555 is powered on and the program is loaded into the FPGA via the platform flash and CPLD. Following the indication given by the ML555 that the algorithm is successfully programed to the Virtex-5, the user pushes the start button (AF20) to initiate the flow of the input signal through the correlation routine. The result is stored in RAM and can be transferred to a computer using the UART interface introduced in Section 4.3. The serial output is the ASCII representation of the hex values. Figure 4.13 provides a visual representation of the output format. Once the ASCII file has been captured

Table 4.2: ML555 Power Measurement Results

IDLE

Name	Description	Pin	Resistance (Ω)	Current (A)	Power (W)
AVTTX	1.2V GTP Termination	P41	0.01	0.00	0.00
AVCCPLL	1.2V PLL Supply	P42	0.01	0.05	0.06
AVCC	1.0V GTP Supply	P43	0.01	0.02	0.02
VCCINT	1.0V FPGA Internal	P44	0.01	0.48	0.48
PCIe Converter	5.0V PCIe Supply	P19	0.15	0.00	0.00
					0.56

RUNNING

Name	Description	Pin	Resistance (Ω)	Current (A)	Power (W)
AVTTX	1.2V GTP Termination	P41	0.01	0.00	0.00
AVCCPLL	1.2V PLL Supply	P42	0.01	0.05	0.06
AVCC	1.0V GTP Supply	P43	0.01	0.02	0.02
VCCINT	1.0V FPGA Internal	P44	0.01	0.58	0.58
PCIe Converter	5.0V PCIe Supply	P19	0.15	0.00	0.00
					0.66

on the computer, MATLAB is used to convert the ASCII characters into a numerical format and generate the plots.

The correlation result provided by the experiment is shown in Figure 4.14a. There is an obvious correlation spike at the beginning of the second template. A comparison to the MATLAB result shows that the correlation peaks occur at the same point in time (Figure 4.14b). However, the peak for the FPGA result does not quite reach the same magnitudes observed for the MATLAB or Simulink correlation.

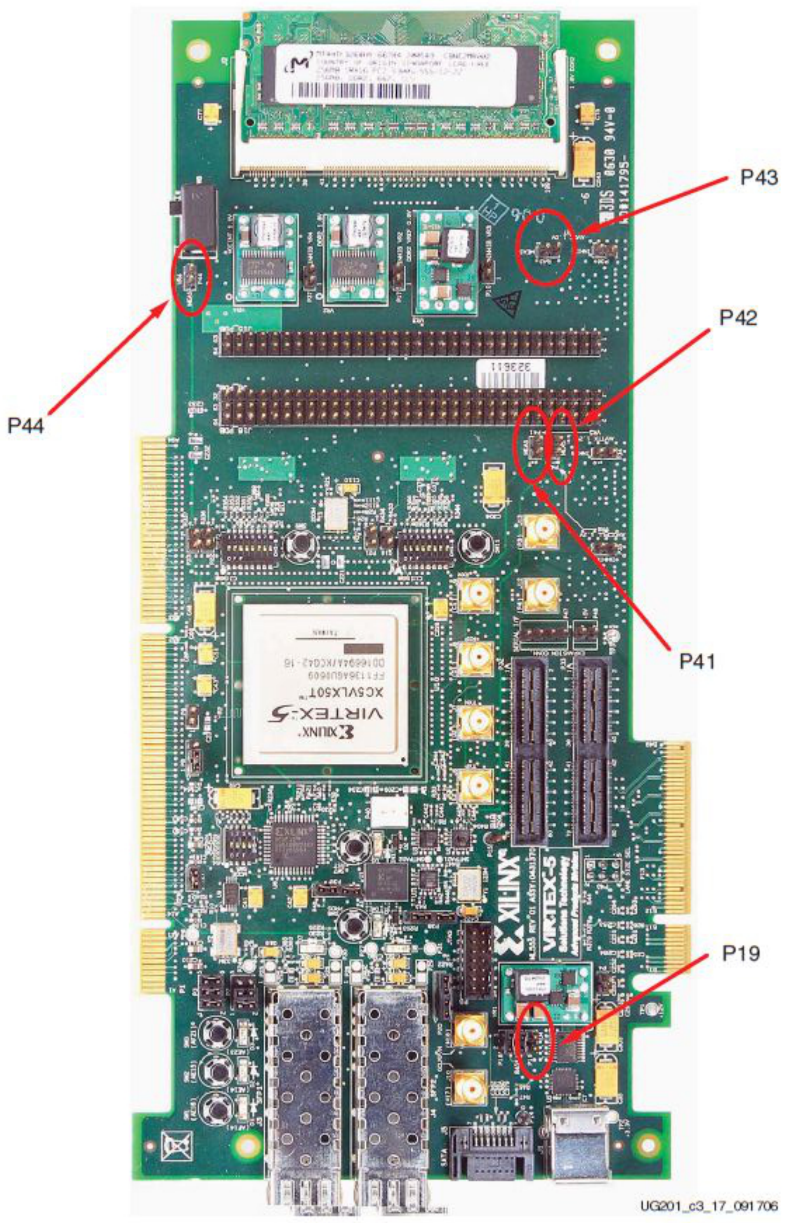


Figure 4.12: ML555 Current Sensing Locations [43]

There are several factors that could contribute to the difference observed in Figure 4.14b. First, the MATLAB and Simulink results are calculated using 64-bit double precision values while the FPGA correlator uses 16-bit. The loss in precision could introduce quantization noise where the filter coefficients are no longer perfectly matched to

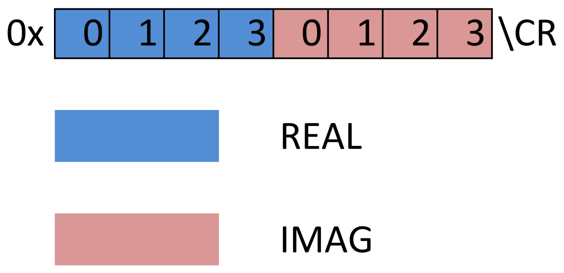


Figure 4.13: FPGA Correlator's Output Format

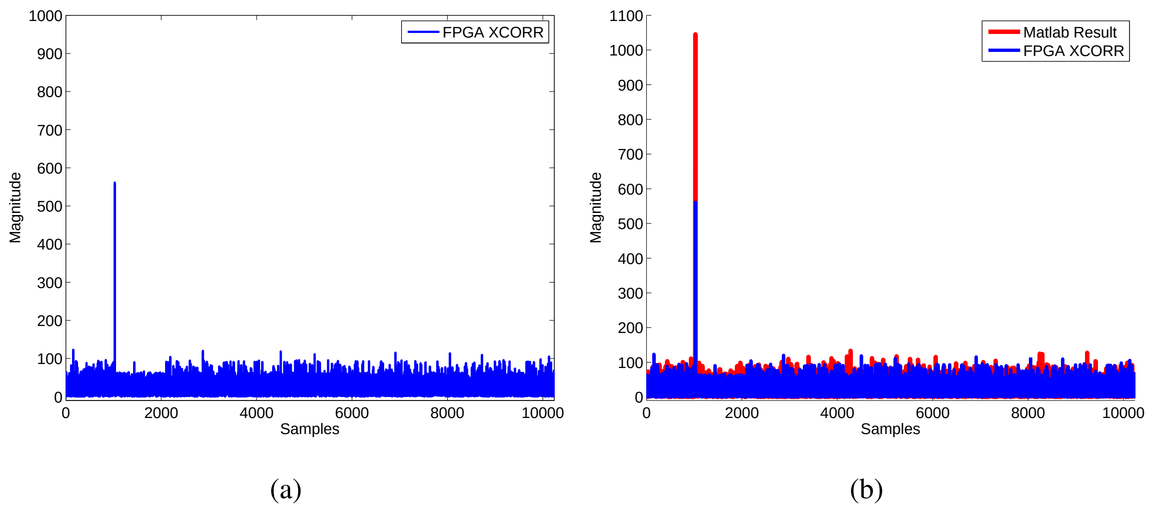


Figure 4.14: Experimental Results (a) FPGA Correlation, (b) FPGA Result Compared to MATLAB

the FFT output. When the FFT results from the FPGA and MATLAB are plotted together, there are obvious differences in the two results (Figure 4.15).

These differences alone account for much of the error observed in Figure 4.14. A projected correlation result could be calculated using the values from the FPGA's FFT. A

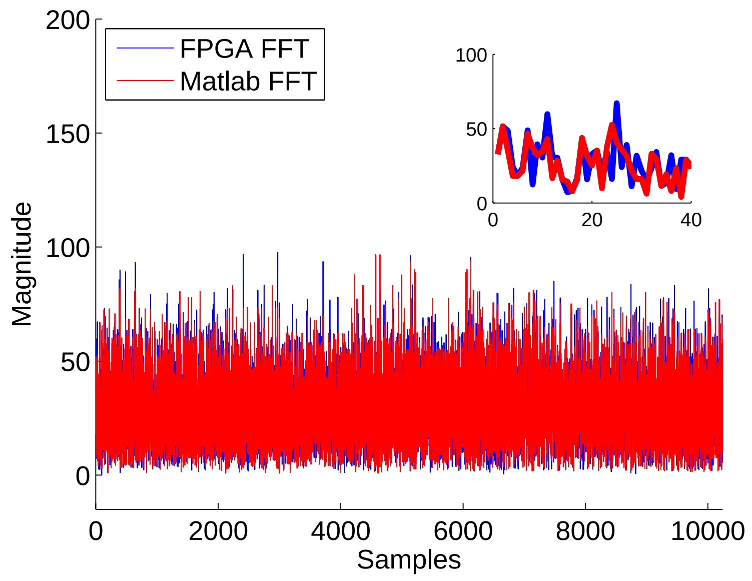


Figure 4.15: FPGA FFT Versus MATLAB FFT (The plot in the upper right-hand corner is a closeup of the first 40 samples)

plot of this projection is shown in Figure 4.16. The peak for the projected correlation does not differ very much from the magnitude observed in Figure 4.14.

Other factors that could cause differences between the hardware and Simulink correlation results are truncation and internal noise. To keep from adding additional components to the correlation design, the output from the FFT and IFFT was simply truncated in order to maintain the 16-bit flow. This truncation results in a biased rounding scheme for the least significant digit. Ideally, the least significant digit would be rounded based on the digit(s) to the right of the truncation (analogous to rounding in base 10). Finally, internal noise sources could result in false logic outputs commonly referred to as bit errors. An erroneous calculation within any of the correlation algorithm's computational elements could result in an output that does not agree with simulation.

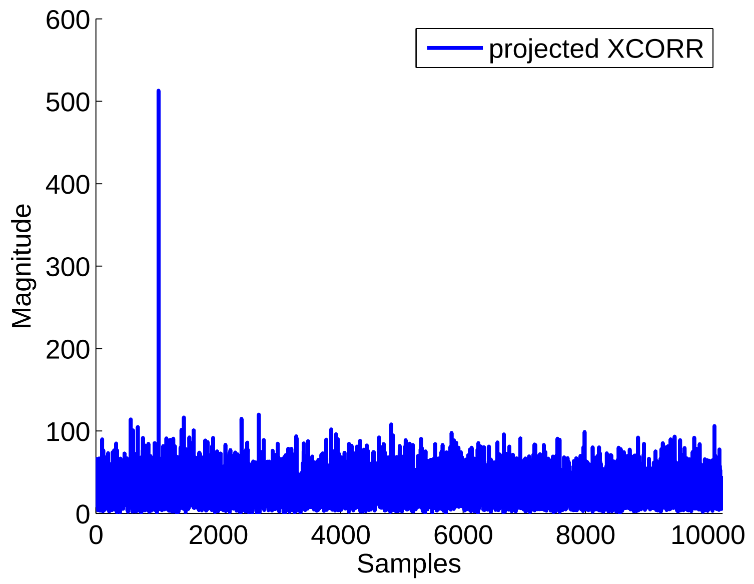


Figure 4.16: Projected Correlation From FPGA FFT Results

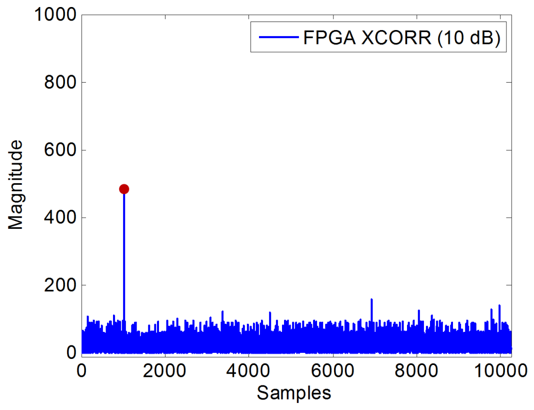
It should be reiterated, however, that despite the minor deviation from theory the FPGA correlation algorithm was able to produce a usable result. The algorithm correctly identified the portion of the input signal that corresponded to the matched filter. One trade-off from the miniaturization effort may be that higher signal to noise ratios are needed to detect the desired signal than in the current AFIT RNR.

4.5 Performance of the Correlation Algorithm in Degrading SNRs

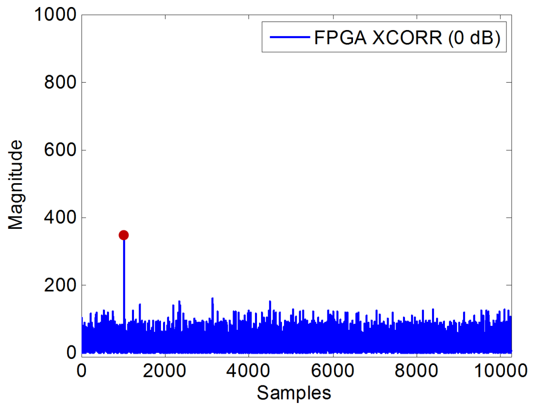
The successful identification of the signal of interest in the previous section naturally leads to the following question: How poor can the SNR of the input signal be and still be detected by the FPGA correlator? To answer this question, normally distributed noise was added to the input signal according to the following formula:

$$\text{InputSignal} = x[n] + \sqrt{\sigma^2} * w[n], \tag{4.1}$$

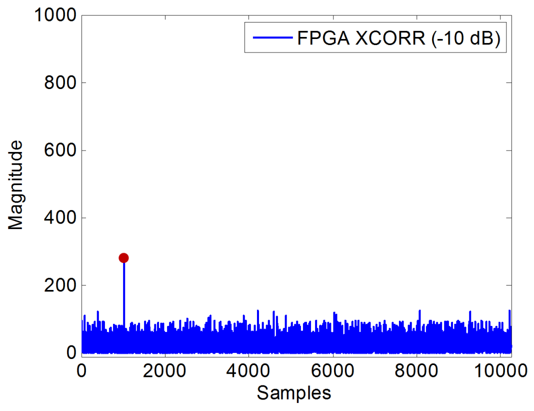
where $x[n]$ is the input signal used in the previous section, σ^2 is the power of the noise signal and $w[n]$ is a normally distributed random signal with a mean of zero and a standard deviation of one. Input signals with SNRs of 10 dB, 0 dB, -10 dB, and -15 dB were generated in MATLAB and then put through the FPGA correlation routine. The plots for each of these experiments are shown in Figure 4.17.



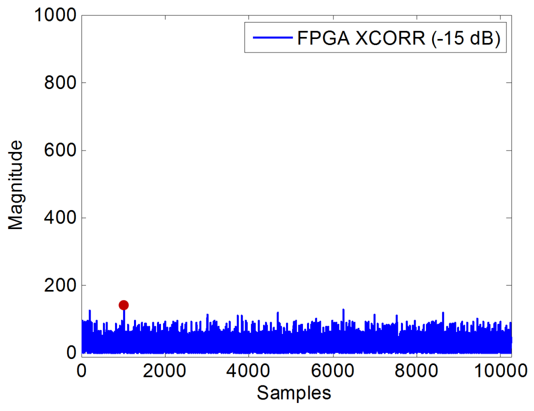
(a)



(b)



(c)



(d)

Figure 4.17: Correlation Results with Varying Input SNRs (a) SNR = 10 dB, (b) SNR = 0 dB, (c) SNR = -10 dB and (d) SNR = -15 dB

The correlation routine correctly identified the signal of interest in each of the four test cases. As the noise conditions of the input signal worsened, the magnitude of the correlation spike decreased. At an SNR of -15 dB the maximum correlation spike was just above the noise floor, and as the SNR approached -20 dB the correlation algorithm was unable to locate the signal of interest within the input data stream.

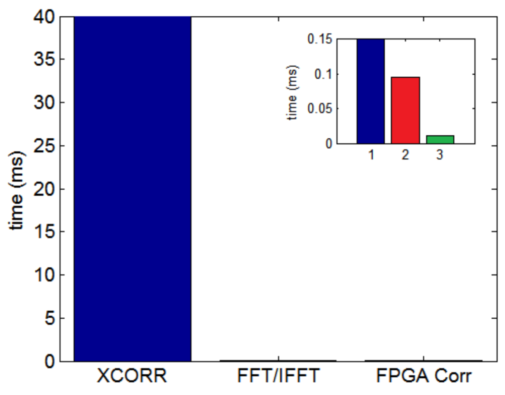
4.6 Timing Results

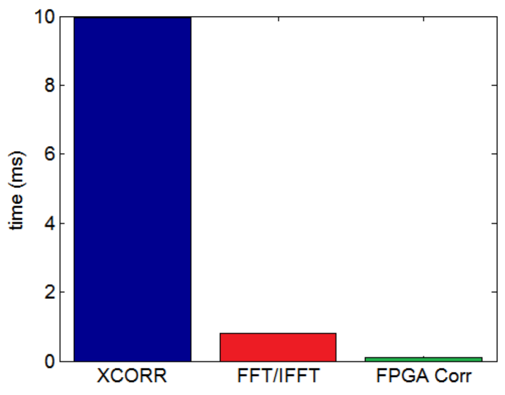
Now that the correlation algorithm has been demonstrated, the computational speed will be compared to similar calculations in MATLAB. To begin, the time required for the FPGA-based algorithm to compute a single 1024-point correlation will be determined. The time will be compared to the computational time required to execute two commonly used MATLAB correlation routines: the xcorr function and an FFT/IFFT approach. Finally, the same comparison will be made for ten consecutive calculations. The matlab routines will be computed on a laptop equipped with an Intel i5 processor and 8 giga-bytes of RAM.

Because the ML555 does not have an IO port that could be easily interfaced with an oscilloscope or a logic analyzer, an alternative method will be used to determine the computational time of the FPGA routine. Included in FPGA correlation algorithm is a 32-bit counter. When the start button on the ML555 is pressed by the user, the counter begins to increment on every clock cycle. The counter will continue to increment until the final correlation data point is written to the FPGA's memory. At the end of the calculation the results are sent to the computer via the UART module. The count can then be multiplied by the clock period to determine the overall time elapsed.

The timing experiment was run ten times to ensure that the results were consistent. On each of the ten trials, the number of clock cycles required to compute the correlation result for the 10,240 input samples was measured at 13,053. When multiplied by the 8 ns clock period, the total time elapsed was 104.42 us. Figure 4.18 shows how the performance of the FPGA algorithm compares to the two MATLAB methods. In MATLAB, the fastest routine was the Fourier transform method of computing the correlation. However, the

FPGA algorithm out performed the fastest MATLAB routine by six times, and was 60 times faster than the xcorr function.





(a) (b)

Figure 4.18: Timing Results (a) 1024-point Correlation, (b) 10240-point Correlation

If the FPGA routine was optimized to run at higher clock speeds, the performance gap between computer-based correlation and the FPGA routine would continue to increase (i.e., a system clock of 200 MHz would decrease the time required to execute the algorithm to 65.27 us). Furthermore, when multiple templates are involved the serial nature of the computer-base calculations would lead to a linear increase in computation time. The FPGA implementation could execute these calculations in parallel and little to no increase in computation time would be observed.

4.7 Conclusion

In this chapter, the proposed correlation algorithm was realized in a computer-base model and in representative hardware. The simulation produced results that were nearly identical to theory, and the FPGA implementation of the routine showed tremendous potential for miniature RNR applications. The FPGA correlator is predicted to use very little power and leaves a significant portion of the Virtex-5 open for additional development. Furthermore, the routine was able to correctly identify the signal of interest, and it out performed computer-based routines in computational speed. The next chapter will review the research objectives and make conclusions based on the observed results.

V. Conclusions

5.1 Chapter Overview

A miniaturized RNR provides a sensor with a unique set of characteristics to a plethora of design possibilities. The smaller system can be infused into applications ranging from collision avoidance in autonomous vehicles to hand-held navigation systems. The intent of this research effort was to advance AFIT's RNR capability while initiating the effort to shrink the current system down to a chip-based architecture. This chapter reviews the stated research goals, presents a summary of the research results and contributions, and identifies areas for future study to further advance the miniaturization of AFIT's RNR.

5.2 Research Goals

The over-arching goal of this research was to begin the exploration into suitable chip-based architectures for the RNR. One of the most significant hurdles to the miniaturization of the noise radar, is reducing the size and power consumption of the radar's components while maintaining the ability to accomplish high speed signal processing. As a result, this drove the primary focus toward identifying a signal processing component that could meet all three of the aforementioned objectives and adequately perform the noise radar's receive function. To demonstrate the suitability of the target hardware, an algorithm had to be developed to execute the correlation of an incoming RF waveform with a delayed copy of the transmit signal. The current AFIT RNR was used as a performance baseline to which the system under test would be compared to.

5.3 Results and Contributions

The requirement for capable signal processors that are compact and energy efficient narrowed the field of possibilities to DSPs and FPGAs. An FPGA-base architecture was chosen because of the performance gains attributed to parallel processing. Furthermore, FPGAs are highly reconfigurable and can be molded to fit the design constraints of future research projects. The latest class of FPGA technology contains many more logic elements than previous generations. The larger FPGAs could possibly eliminate multi-chip architectures saving a tremendous amount of space and energy consumption.

The selection of the FPGA as the primary signal processing device had a profound impact on the algorithm used for computing the cross-correlation. The algorithm had to maximize the number of parallel computations while minimizing the impact on the FPGA's resources. For this reason, the R2^2SDF FFT was chosen as the primary computational element in the correlation routine. The mathematical and architectural development of the routine was presented.

To ensure that the proposed correlation routine worked as expected, a Simulink model was developed. Simulations of the correlation model showed little to no difference when compared to the same computations in MATLAB.

The successful verification of the correlation model led to an FPGA implementation of the design. The design of the correlation algorithm closely followed the setup proposed in the model so that the two results could be compared. Post-implementation reports of the FPGA-based correlator indicated zero timing failures, less than a Watt of power consumption, and a 44% utilization of the Virtex-5's logic resources.

The experimental results from the hardware implementation of the correlation routine were promising. The algorithm was able to successfully locate the presence of a signal of interest within an input data stream, and despite a system clock of only 125 MHz, the algorithm was able to compute the correlation six times faster than MATLAB (60 times faster than the xcorr function). The contributions provided by this research effort has culminated in a foundation upon which the miniaturized RNR's receiver can be built.

5.4 Future Work

While successes where realized in this research effort, the immensity of the miniaturization effort has left much of the work to follow-on efforts. These efforts have been sorted into three categories: hardware design work, algorithm improvements, and ancillary applications.

5.4.1 Hardware Design Work.

For a prototype chip-based RNR to become a reality, there is some hardware design left to accomplish. For starters a target platform needs to be identified so that the following questions can be answered: What inputs will the host vehicle provide to the RNR? What outputs should the RNR provide to the host vehicle and what does the update rate need to be? What are the minimum and maximum power levels available to the RNR? What are the weight and space limitations? What will the primary mode of operation be for the RNR sensor: communication or radar? What range is needed to achieve each mode of operation? What is the average range/battery life of the host vehicle without the RNR? Once these questions have been answered the systems engineering approach can be applied to further refine the RNR's design. In addition to refining the hardware design, the following research topics could provide some valuable insight:

- An implementation of the correlation algorithm in the latest generation of FPGAs (i.e. the Virtex 7) should be explored to determine its performance and scalability.
- A high-speed FPGA-based transmitter should be explored to determine feasibility of a pseudo-random architecture.
- Interface devices like ultra-high speed data converters, memory units should be integrated with the FPGA and the requisite signal processing routines should be tested.

- Ultimately, the integration of the miniature RNR with the host vehicle will need to be demonstrated.

5.4.2 Algorithm Improvements.

Research has provided the following areas where the correlation routine can be improved:

- Designing a DIT implementation of the R2^2SDF IFFT would cut the latency of the current correlation routine in half and would lead to a reduction in the amount of FPGA resources consumed by the design.

- An FPGA-based signal acquisition routine should be developed so that the nodes of a distributed network could synchronize the correlation routine with the incoming signal.

- A systolic, pipelined approach to the FFT and IFFT should be explored to determine if real-time correlation can be accomplished.

- Direct correlation receivers produce an enormous amount of data and could benefit from the emerging field of compressed sampling, and the exploration of hardware designs that exploit this theory could achieve better performance at lower data rates (see Appendix C).

5.4.3 Ancillary Applications.

In addition to the chip-based RNR other opportunities exists where the research presented in this thesis can be applied:

- The miniaturized RNR can be reconfigured to a highly portable, secure communication device.

- An FPGA can be added to AFIT's current RNR and many of the computationally intensive processes can be off-loaded to FPGA-based signal processing algorithms, and additional logic can be added to synchronize a distributed network of noise radars for communication or navigation routines.

Appendix A: HDL Correlator

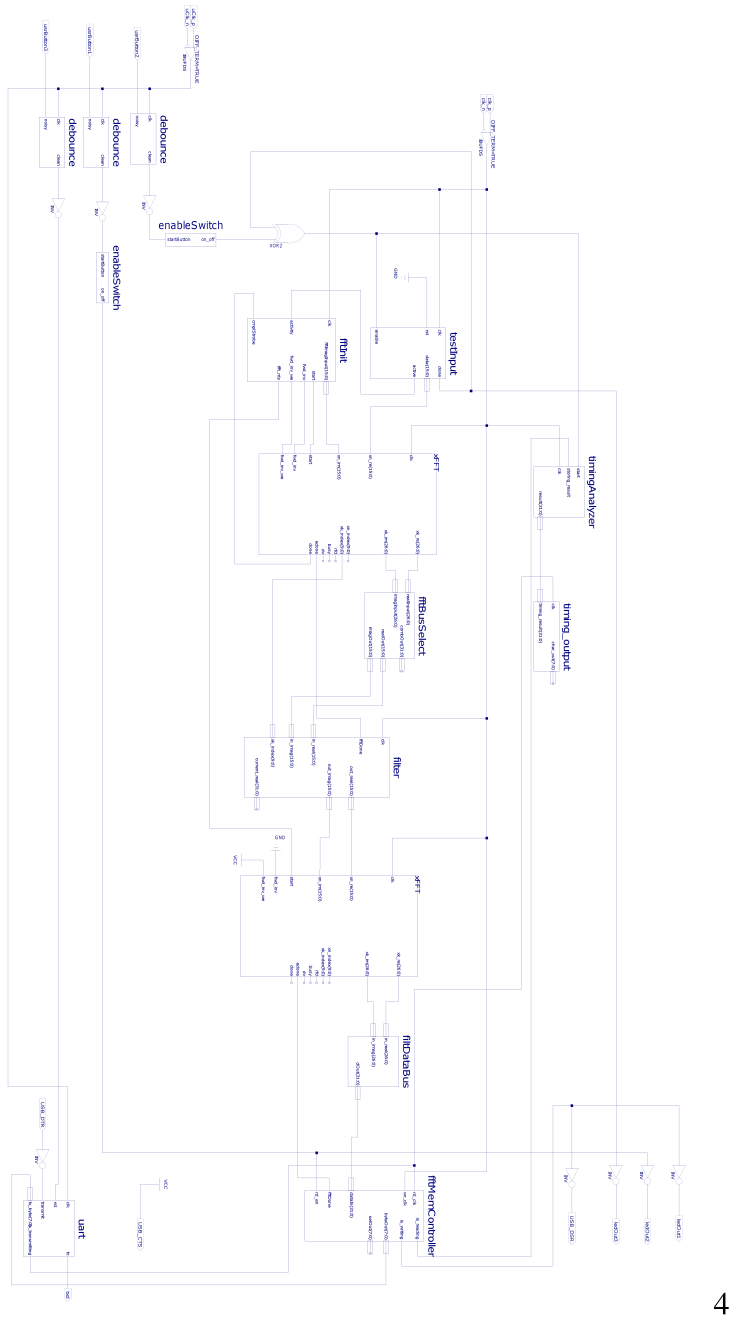


Figure A.1: FPGA Correlator Schematic

```
//////////////////////////////////////////////////////////////////
// Configurable delay element used to align the FFT output and
// the filter data
//
// Aaron Myers
// Winter 2012
//////////////////////////////////////////////////////////////////
module circBuffer (clk, reset, enable, wrdata, rddata) ;

  input clk ; // clock
  input reset ; // asynchronous reset
  input enable ; // assert high to write data
  input [15:0] wrdata ; // data to write port - use 1 byte width for this example
  output [15:0] rddata ; // data on read port

  parameter LAST_ELEMENT = 4'hf;
  parameter WR_START = 0;

  // locally used flops
  reg [3:0] wrptr = WR_START;
  reg [3:0] rdptr = 0; // write and read pointers
  wire [3:0] new_wrptr;
  wire [3:0] new_rdptr;
  reg wrenable = 1 ;
  reg rdenable = 0;  // write enable to memory

  // basic 2 port memory – for this small size, assume read is asynch
  dualPortRAM dpr1(
  .clka(clk),
  .wea(wrenable),
  .ena(enable),
  .addra(wrptr),
  .dina(wrdata),
  .douta(),
  .clkb(clk),
  .web(rdenable),
  .addrb(rdptr),
  .dinb(),
  .doutb(rddata));


  assign new_wrptr = (wrptr == LAST_ELEMENT) ? 2'b0 : (wrptr + 1);
  assign new_rdptr = (rdptr == LAST_ELEMENT) ? 2'b0 : (rdptr + 1);

  always @ (posedge clk)

    if (reset) begin // async reset
       wrptr <= WR_START ;
       rdptr <= 0 ;
    end
    else begin
      wrptr <= new_wrptr ;
      rdptr <= new_rdptr ;
    end

endmodule
```

```
//-----------------------------------------------------------------------------
//Design Name: counter
//File Name: counter.v
//Function: toggles an led everytime the counter rolls over.
//Coder: Aaron Myers
//-----------------------------------------------------------------------------
module counter(clk, enable, outPulse, dataOut_r, dataOut_i, addrOut);

//input ports
input clk;
input enable;

//output ports
output outPulse;
output [26:0] dataOut_r;
output [26:0] dataOut_i;
output [9:0] addrOut;

//parameters
parameter TARGET = 10'h3FF;
parameter EDONE = 10'h3FE;

//data types
reg[26:0] count = 0;
reg [9:0] pulseCount = 0;
reg [9:0] addrCount = 0;
reg [9:0] numPulses = 0;

//assign statements
assign outPulse = pulseCount == EDONE;
assign dataOut_r[26:0] = count[26:0];
assign dataOut_i[26:0] = count[26:0];
assign addrOut[9:0] = addrCount[9:0];

//--------------program---------------
always @(posedge clk) begin

    if (enable) begin
        pulseCount <= pulseCount + 1;
        addrCount <= addrCount + 1;

        if (pulseCount == TARGET) begin
            numPulses = numPulses + 1;
        end

        if(numPulses >= 1) begin
            count = count + 1;
        end
    end

end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module
//
// Source taken from: http://web.mit.edu/6.111/www/f2005/code/jtag2mem_6111/debounce.v.html
////////////////////////////////////////////////////////////////////////////////

module debounce (clk, noisy, clean);
   input clk, noisy;
   output clean;

   parameter NDELAY = 2500000;
   parameter NBITS = 23;

   reg [NBITS-1:0] count;
   reg xnew, clean;

   always @(posedge clk)
     if (noisy != xnew) begin xnew <= noisy; count <= 0; end
     else if (count == NDELAY) clean <= xnew;
     else count <= count+1;

endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Simple enable switch for the input signal
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module enableSwitch(startButton, on_off);

input startButton;
output on_off;

parameter ON = 1'b1;
parameter OFF = 1'b0;

reg sw_enable = 0;

assign on_off = sw_enable;

always @(posedge startButton) begin
    case(sw_enable)
        OFF: sw_enable = ON;
        ON: sw_enable = OFF;
    endcase

end


endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Simply combines the two 16-bit real and imag values into a single 32 bit word.
//
// Aaron Myers
// Winter 2012
//
////////////////////////////////////////////////////////////////////////////////
module fftBusSelect(realInput, imagInput, realOut, imagOut, combOut);

//define I/O
input [26:0] realInput;
input [26:0] imagInput;
output [15:0] realOut;
output [15:0] imagOut;
output [31:0] combOut;

//define elements

//program code
assign realOut[15:0] = realInput[18:3];
assign imagOut[15:0] = imagInput[18:3];
assign combOut[31:0] =  {realInput[18:3], imagInput[18:3]};

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module handles all of the control parameters for the first fft instance
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module fftInit(clk, activity, cmplStrobe, fftImagInput, start, fwd_inv, fwd_inv_we, ifft_rdy);

//define inputs and outputs
input clk;
input activity;
input cmplStrobe;
output [15:0] fftImagInput;
output fwd_inv;
output fwd_inv_we;
output ifft_rdy;
output start;

//define parameters
parameter FFT_TYPE = 1'b1;
parameter BEGIN = 1'b1;
parameter STOP = 1'b0;
parameter MULT_LATENCY = 8;

//define elements
reg [3:0] fftCount = 0;
reg [4:0] multLatencyCount = MULT_LATENCY;
reg ifftStatus = STOP;

//make assignments
assign fwd_inv = FFT_TYPE;
assign fftImagInput[15:0] = 16'h0;
assign fwd_inv_we = 1'b0;
assign start = activity;
assign ifft_rdy = ifftStatus == BEGIN;


//program starts here
always @(posedge clk) begin

    if (cmplStrobe) begin
        fftCount = fftCount + 1;
    end

    if(fftCount == 1) begin
        if(multLatencyCount == 0) begin
            ifftStatus = BEGIN;
        end
        else begin
            multLatencyCount = multLatencyCount - 1;
        end
    end

end




endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// This module will serve as the memery controller for the xcorr results.
//
// Aaron Myers
// Winter 2012
//
////////////////////////////////////////////////////////////////////////////////////
module fftMemController(dataIn, fftDone, rd_en, wr_clk, rd_clk, is_reading, is_writing, byteOut,
 selOut);

//define I/O pins
input [31:0] dataIn;
input fftDone;
input rd_en;
input rd_clk;
input wr_clk;
output is_reading;
output is_writing;
output [7:0] byteOut;
output [7:0] selOut;

//define parameters
parameter ERROR = 1'b1;
parameter NFFT = 1024;
parameter NUMELEMENTS = 10240;
parameter WRITE = 1'b1;
parameter READ = 1'b0;
parameter START = 1'b1;
parameter STOP = 1'b0;
parameter ASCII_NUM = 48;
parameter ASCII_LET = 55;

//define elements
reg [31:0] current_result;
reg [13:0] totalWrites = 0;
reg [13:0] addr = 0;
reg [3:0] fftCount = 0;
reg [7:0] sel = 0;
reg [7:0] outReg = 0;
reg [3:0] char = 0;
reg enable = 0;
reg wrStatus = 1;
reg clkSelect = 0;
wire [13:0] ramAddr;
//wire [31:0] wr_data;
wire [31:0] ramDataOut;
wire wrEnable;
wire clk;

//assignment statements
//assign wr_data[31:0] = dataIn[31:0];
assign byteOut[7:0] = outReg[7:0]; //element count will be provided to the cpu via uart
assign selOut[7:0] = sel[7:0];
assign wrEnable = wrStatus == WRITE;
assign ramAddr[13:0] = addr[13:0];
assign is_reading = wrStatus == WRITE;
assign is_writing = (wrStatus == READ) & (enable==START) & rd_en;
//assign clk = (wrStatus == READ) ? rd_clk : wr_clk;

//instatiation
resultRAM rRAM0(
.clka(clk),
.wea(wrEnable),
.addra(ramAddr),
.dina(dataIn),
.douta(ramDataOut));

BUFGMUX_CTRL clkbuf1(
.I0(wr_clk),
.I1(rd_clk),
.S(clkSelect),
.O(clk));

//program starts here
always @(posedge clk) begin

    current_result <= ramDataOut;

    if(fftDone) begin
        fftCount = fftCount + 1;
    end

    if(fftCount > 0 && totalWrites == 0) begin
        enable = START;
    end

    if(enable) begin

        case(wrStatus)
        WRITE: begin
            if (totalWrites == 0) begin
                addr = 0;
                totalWrites = totalWrites + 1;
            end
            else if (totalWrites != 0 && totalWrites < NUMELEMENTS) begin
                addr = addr + 1;
                totalWrites = totalWrites + 1;
            end else begin
                addr = 0;
                clkSelect = 1;
                wrStatus = READ;
            end
        end

        //--------------------------------------------------------
        // Need to send the 32-bit data to an external UART trans-
        // mitter, therefore this module will send the 32-bit complex
        // result out in the following format: Rh, Rl, Ih, Il.
        //-------------------------------------------------------
        READ: begin
            if (rd_en) begin

                case(sel)
                    8'h0: begin
                        char = current_result[31:28];
                        sel = sel+1;
                    end
                    8'h1: begin
                        char = current_result[27:24];
                        sel = sel+1;
                    end
                    8'h2: begin
                        char = current_result[23:20];
                        sel = sel+1;
                    end
                    8'h3: begin
                        char = current_result[19:16];
                        sel = sel+1;
                    end
                    8'h4: begin
                        char = current_result[15:12];
                        sel = sel+1;
                    end
                    8'h5: begin
                        char = current_result[11:8];
                        sel = sel+1;
                    end
                    8'h6: begin
                        char = current_result[7:4];
                        sel = sel+1;
                    end
                    8'h7: begin
                        char = current_result[3:0];
                        sel = sel+1;
                    end
                    8'h8: begin
                        outReg = 8'hff;
                        addr = addr + 1;
                        sel = 0;
                    end
                    default: sel = 0;
                endcase

                if (outReg == 8'hff && sel ==0) begin
                    outReg = 8'h0d;
                end
                else if(char <= 9) begin
                    outReg = char + ASCII_NUM;
                end
                else begin
                    outReg = char + ASCII_LET;
                end

                if(addr >= totalWrites) begin
                    enable = STOP;
                end
            end

        end

        default: wrStatus = WRITE;

        endcase

    end

end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Selects the 16 bits for the IFFT from the 32-bit filter output
//
// Aaron Myers
// Winter 2012
//////////////////////////////////////////////////////////////////////////////////
module filtDataBus(in_real, in_imag, dOut);

input [26:0] in_real;
input [26:0] in_imag;
output [31:0] dOut;

assign dOut = {in_real[26:11], in_imag[26:11]};

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module contains two instances of single port roms that contain the coefficents
// for the correlator's matched filter and an instance of a complex multiplier.
// The filter corresponds to a "template" matched to the second 1024 pt sample in the
// input signal.
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module filter(in_real, in_imag, xk_index, fftDone, clk, out_real, out_imag, current_real);

//define inputs and outputs
input clk;
input fftDone;
input [9:0] xk_index;
input [15:0] in_real;
input [15:0] in_imag;
output reg [15:0] out_imag;
output reg [15:0] out_real;
output reg [31:0] current_real;

//define element
reg start = 0;
reg [3:0] fftCount = 0;
reg [9:0] rd_addr = 0;
wire [15:0] filtImag;
wire [15:0] filtReal;
wire [9:0] memAddr;
wire [15:0] delayed_out_r;
wire [15:0] delayed_out_i;
wire [15:0] delayed_filt_r;
wire [15:0] delayed_filt_i;
wire [15:0] outImag;
wire [15:0] outReal;
wire carry = 1'b0;
wire rst = 1'b0;
wire enable;
wire wea = 1'b0;



//assign statements
//assign current_real[31:0] = {delayed_out_r[15:0], delayed_out_i[15:0]};
assign memAddr[9:0] = rd_addr[9:0];
assign enable = start == 1'b1;

//instantiation
ti_ROM fI(
.clka(clk),
.wea(wea),
.addra(memAddr),
.douta(filtImag));

tr_ROM fR(
.clka(clk),
.wea(wea),
.addra(memAddr),
.douta(filtReal));

circBuffer#(.LAST_ELEMENT(8'hf), .WR_START(1)) filtbuff_real(
.clk(clk),
.reset(rst),
.enable(enable),
.wrdata(filtReal),
.rddata(delayed_filt_r)) ;

circBuffer#(.LAST_ELEMENT(8'hf), .WR_START(1)) filtbuff_imag(
.clk(clk),
.reset(rst),
.enable(enable),
.wrdata(filtImag),
.rddata(delayed_filt_i)) ;

circBuffer#(.LAST_ELEMENT(8'hf), .WR_START(2)) delaybuff_real(
.clk(clk),
.reset(rst),
.enable(enable),
.wrdata(in_real),
.rddata(delayed_out_r)) ;

circBuffer#(.LAST_ELEMENT(8'hf), .WR_START(2)) delaybuff_imag(
.clk(clk),
.reset(rst),
.enable(enable),
.wrdata(in_imag),
.rddata(delayed_out_i)) ;

ComplexMult cm1(
.clk(clk),
.round_cy(carry),
.ai(delayed_out_i),
.bi(delayed_filt_i),
.ar(delayed_out_r),
.br(delayed_filt_r),
.pi(outImag),
.pr(outReal));

//Program Code...
always @(posedge clk) begin

    current_real[31:0] = {outReal[15:0], outImag[15:0]};
    out_real = outReal;
    out_imag = outImag;
    if (fftDone) begin
        fftCount <= fftCount + 1;
        start <= 1'b1;
    end

    if (start) begin
        rd_addr <= rd_addr + 1;
    end

end

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module handles all of the control parameters for the second fft instance
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module ifftInit(clk, enable, start, fwd_inv, fwd_inv_we);

//define inputs and outputs
input clk;
input enable;
output fwd_inv;
output fwd_inv_we;
output start;

//define parameters
//parameter SCALE_SCH = 10'b1010101010; //divides end result by 1024
parameter FFT_TYPE = 1'b0;
parameter BEGIN = 1'b1;
parameter STOP = 1'b0;

//define elements
//reg [9:0] scale_sch = SCALE_SCH;
reg enableStatus = STOP;

//make assignments
assign fwd_inv = 1'b0;
assign fft_fwd_inv_we = 1'b1;
assign start = (enableStatus == BEGIN) ? 1'b1: 1'b0;


//program starts here
always @(posedge clk) begin

    if(enable) begin
        enableStatus <= BEGIN;
    end
    else begin
        enableStatus <= STOP;
    end


end


endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////
// This module will serve as the memery controller for the input2MemTest.
//
// Aaron Myers
// Winter 2012
//
////////////////////////////////////////////////////////////////////////////////////
module input2MemController(dataIn, data_rdy, rd_en, wr_clk, rd_clk, is_reading, is_writing,
byteOut);

//define I/O pins
input [15:0] dataIn;
input data_rdy;
input rd_en;
input rd_clk;
input wr_clk;
output is_reading;
output is_writing;
output [7:0] byteOut;

//define parameters
parameter NUMELEMENTS = 10240;
parameter WRITE = 1'b1;
parameter READ = 1'b0;
parameter START = 1'b1;
parameter STOP = 1'b0;
parameter ASCII_NUM = 48;
parameter ASCII_LET = 55;

//define elements
reg [13:0] totalWrites = 0;
reg [13:0] addr = 0;
reg [7:0] sel = 0;
reg [7:0] outReg = 0;
reg [3:0] char = 0;
reg clkSelect = 0;
reg enable = 0;
reg rw_status = 1;
wire [13:0] ramAddr;
wire [31:0] ramDataIn;
wire [31:0] ramDataOut;
wire wrEnable;
wire rw_select;
wire clk;

//assignment statements
assign byteOut[7:0] = outReg[7:0]; //element count will be provided to the cpu via uart
assign ramAddr[13:0] = addr[13:0];
assign ramDataIn [31:0] = {16'h0, dataIn};
assign rw_select = rw_status == WRITE;
assign is_reading = rw_status == WRITE;
assign is_writing = (rw_status == READ) & (enable == START);
```

```
//instatiation
resultRAM in2mem(
.clka(clk),
.wea(rw_select),
.addra(ramAddr),
.dina(ramDataIn),
.douta(ramDataOut));

BUFGMUX_CTRL clkbuf1(
.I0(wr_clk),
.I1(rd_clk),
.S(clkSelect),
.O(clk));

//program starts here
always @(posedge clk) begin
    if (data_rdy == 1 && totalWrites == 0) begin
        enable = START;
    end

    if(enable == START) begin

        case(rw_status)
            WRITE: begin
                if (totalWrites == 0) begin
                    addr = 0;
                    totalWrites = totalWrites + 1;
                end
                else if (totalWrites != 0 && totalWrites < NUMELEMENTS) begin
                    addr = addr + 1;
                    totalWrites = totalWrites + 1;
                end else begin
                    addr = 0;
                    rw_status = READ;
                end
            end

            //-------------------------------------------------------
            // Need to send the 32-bit data to an external UART trans-
            // mitter, therefore this module will send the 32-bit complex
            // result out in the following format: Rh, Rl, Ih, Il.
            //-------------------------------------------------------
            READ: begin
                clkSelect = 1;
                if (rd_en) begin
                    case(sel)
                        8'h0: begin
                            char = ramDataOut[31:28];
                            sel = sel+1;
                        end
                        8'h1: begin
                            char = ramDataOut[27:24];
                            sel = sel+1;
                        end
                        8'h2: begin
                            char = ramDataOut[23:20];
                            sel = sel+1;
                        end
                        8'h3: begin
                            char = ramDataOut[19:16];
                            sel = sel+1;
                        end
                        8'h4: begin
                            char = ramDataOut[15:12];
                            sel = sel+1;
                        end
                        8'h5: begin
                            char = ramDataOut[11:8];
                            sel = sel+1;
                        end
                        8'h6: begin
                            char = ramDataOut[7:4];
                            sel = sel+1;
                        end
                        8'h7: begin
                            char = ramDataOut[3:0];
                            sel = sel+1;
                        end
                        8'h8: begin
                            outReg = 8'hff;
                            addr = addr + 1;
                            sel = 0;
                        end
                    endcase

                    if (outReg == 8'hff) begin
                        outReg = 8'h0d;
                    end
                    else if(char <= 9) begin
                        outReg = char + ASCII_NUM;
                    end
                    else begin
                        outReg = char + ASCII_LET;
                    end

                    if(addr >= totalWrites) begin
                        enable = STOP;
                    end
                end
            end

        endcase
    end

end
```

```
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Checks the input signal.
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module input2UART(dataIn, clk, input_active, rdy_for_next, byteOut, selOut);

//define I/O pins
input [15:0] dataIn;
input clk;
input input_active;
output rdy_for_next;
output [7:0] byteOut;
output [7:0] selOut;

//define parameters
parameter START = 1'b1;
parameter STOP = 1'b0;
parameter ASCII_NUM = 48;
parameter ASCII_LET = 55;

//define elements
reg [7:0] sel = 8'h0;
reg [7:0] outReg = 0;
reg [3:0] char = 0;
reg enable = 0;
wire[15:0] txData;

//assignment statements
assign byteOut[7:0] = outReg[7:0]; //element count will be provided to the cpu via uart
assign rdy_for_next = enable;
assign txData[15:0] = dataIn[15:0];
assign selOut[7:0] = sel[7:0];

//instatiation
//program starts here
always @(posedge clk) begin

        //-------------------------------------------------------
        // Need to send the 32-bit data to an external UART trans-
        // mitter, therefore this module will send the 32-bit complex
        // result out in the following format: Rh, Rl, Ih, Il.
        //-------------------------------------------------------
        case(sel)
            8'h0: begin
                char = txData[15:12];
                sel = 8'h1;
            end
            8'h1: begin
                char = txData[11:8];
                sel = 8'h2;
            end
            8'h2: begin
                char = txData[7:4];
                sel = 8'h3;
            end
            8'h3: begin
                char = txData[3:0];
                sel = 8'h4;
            end
            8'h4: begin
                outReg = 8'hff;
                enable = 1;
                sel = 8'h0;
            end
        endcase

        if (outReg == 8'hff) begin
            outReg = 8'h0d;
        end
        else if(char <= 9) begin
            outReg = char + ASCII_NUM;
            enable = 0;
        end
        else begin
            outReg = char + ASCII_LET;
            enable = 0;
        end


end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module instatiates an async-FIFO and handles I/O control.  Designed to
// handle the uart communication for the FPGACorrelator.
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module outputBuffControler(rst, wr_clk, wr_en, din, rd_clk, dout, uart_en, rd_rdy,
fifo_underflow);


//define I/O pins
input rst;
input wr_clk;
input wr_en;
input [7:0] din;
input rd_clk;
output [7:0] dout;
output uart_en;
output rd_rdy;
output fifo_underflow;

//define parameters
parameter FULL = 1'b1;
parameter READ = 1'b1;
parameter EMPTY = 1'b1;
parameter ENABLE = 1'b0;
parameter DISABLE = 1'b1;

//define elements
reg wr_full = 0;
reg rd_status = 0;
reg en_status = 1;
reg full_status = 0;
wire [7:0] outByte;
wire fifoRd;
wire isempty;
wire isfull;

//assignment definitions
assign fifoRd = rd_status == READ;
assign dout[7:0] = outByte[7:0];
assign uart_en = en_status;
assign rd_rdy = full_status;

//instantiation
outputFIFO outbuff1(
.rst(rst),
.wr_clk(wr_clk),
.din(din),
.wr_en(wr_en),
.full(),
.prog_full(isfull),
.rd_clk(rd_clk),
.dout(outByte),
.rd_en(fifoRd),
.empty(isempty),
.underflow(fifo_underflow));

//program code
always @(posedge wr_clk) begin

    if(isfull) begin
        full_status = FULL;
    end

    if(full_status == FULL) begin
        rd_status = READ;
        en_status = ENABLE;
    end

    if(rd_status == READ) begin
        if (isempty) begin
            rd_status = 0;
            en_status = DISABLE;
            full_status = 0;
        end
    end

end



endmodule
```

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Simply combines the two 16-bit real and imag values into a single 32 bit word.
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module realImagCombiner(realInput, imagInput, combOut);

//define I/O
input [15:0] realInput;
input [15:0] imagInput;
output [31:0] combOut;

//define elements

//assignment statements
assign combOut[31:0] = {realInput[15:0], imagInput[15:0]};

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module will serve as the memery controller for the xcorr results.
//
// Aaron Myers
// Winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module resultMemController(dataIn, xkIndex, fftDone, wr_clk, rd_clk, is_reading, is_writing,
byteOut, selOut);

//define I/O pins
input [31:0] dataIn;
input [9:0] xkIndex;
input fftDone;
input rd_clk;
input wr_clk;
output is_reading;
output is_writing;
output [7:0] byteOut;
output [7:0] selOut;

//define parameters
parameter ERROR = 1'b1;
parameter NFFT = 1024;
parameter NUMELEMENTS = 10240;
parameter WRITE = 1'b1;
parameter READ = 1'b0;
parameter START = 1'b1;
parameter STOP = 1'b0;
parameter ASCII_NUM = 48;
parameter ASCII_LET = 55;

//define elements
reg [13:0] totalWrites = 0;
reg [13:0] addr = 0;
wire [31:0] ramDataOut;
reg [3:0] fftCount = 0;
reg [7:0] sel = 0;
reg [7:0] outReg = 0;
reg [3:0] char = 0;
reg enable = 0;
reg wrStatus = 1;
reg clkSelect = 0;
wire [13:0] ramAddr;
wire wrEnable;
wire clk;

//assignment statements
assign byteOut[7:0] = outReg[7:0]; //element count will be provided to the cpu via uart
assign selOut[7:0] = sel[7:0];
assign wrEnable = wrStatus == WRITE;
assign ramAddr[13:0] = addr[13:0];
assign is_reading = wrStatus == WRITE;
assign is_writing = (wrStatus == READ) & (enable==START);

//instatiation
resultRAM rRAM2(
.clka(clk),
.wea(wrEnable),
.addra(ramAddr),
.dina(dataIn),
.douta(ramDataOut));

BUFGMUX_CTRL clkbuf1(
.I0(wr_clk),
.I1(rd_clk),
.S(clkSelect),
.O(clk));

//program starts here
always @(posedge clk) begin

    if(fftDone) begin
        fftCount = fftCount + 1;
    end

    if(fftCount > 0 && totalWrites == 0) begin
        enable = START;
    end

    if(enable) begin

        case(wrStatus)
        WRITE: begin
            if (totalWrites < NUMELEMENTS) begin
                addr = addr + 1;
                totalWrites = totalWrites + 1;
            end else begin
                addr = 0;
                clkSelect = 1;
                wrStatus = READ;
            end
        end

        //-------------------------------------------------------
        // Need to send the 32-bit data to an external UART trans-
        // mitter, therefore this module will send the 32-bit complex
        // result out in the following format: Rh, Rl, Ih, Il.
        //-------------------------------------------------------
        READ: begin
            case(sel)
                8'h0: begin
                    char = ramDataOut[31:28];
                    sel = sel+1;
                end
                8'h1: begin
                    char = ramDataOut[27:24];
                    sel = sel+1;
                end
                8'h2: begin
                    char = ramDataOut[23:20];
                    sel = sel+1;
                end
                8'h3: begin
                    char = ramDataOut[19:16];
                    sel = sel+1;
                end
                8'h4: begin
                    char = ramDataOut[15:12];
                    sel = sel+1;
                end
                8'h5: begin
                    char = ramDataOut[11:8];
                    sel = sel+1;
                end
                8'h6: begin
                    char = ramDataOut[7:4];
                    sel = sel+1;
                end
                8'h7: begin
                    char = ramDataOut[3:0];
                    sel = sel+1;
                end
                8'h8: begin
                    outReg = 8'hff;
                    addr = addr + 1;
                    sel = 0;
                end
            endcase

            if (outReg == 8'hff) begin
                outReg = 8'h0d;
            end
            else if(char <= 9) begin
                outReg = char + ASCII_NUM;
            end
            else begin
                outReg = char + ASCII_LET;
            end

            if(addr >= totalWrites) begin
                enable = STOP;
            end

        end

        endcase
```

```
    end

end


endmodule
```

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////
// This module will generate a simulated input signal into the fpga correlator.
// Relys on a xilinx ip core for the single port rom to save generating 10000 inputs
// in a case statement.  The signal was generated in matlab as 16-bit, signed  integers
// and follow a normal distribution with zero mean and sigma = 1
//
// Aaron Myers
// Winter 2012
/////////////////////////////////////////////////////////////////////////////////
module testInput(clk, rst, enable, done, active, data);

//Define input and output
input clk;
input rst;
input enable;
output done;
output active;
output [15:0] data;

//Variable definitions
parameter NUMELEMENTS = 10240;
parameter DONE = 1'b1;
parameter WRITE = 1'b0;

//Element definition
reg[14:0] addr = 0;
wire[15:0] dout;
reg wrStatus = WRITE;
wire wrEnable;
wire[14:0] writeAddr;

//Assigment statements
assign data[15:0] = dout[15:0];
assign writeAddr[14:0] = addr[14:0];
assign done = wrStatus == DONE;
assign wrEnable = (wrStatus != DONE) & enable;
assign active = (wrStatus != DONE) & enable;

innputROM d_in(
.clka(clk),
.ena(wrEnable),
.addra(writeAddr),
.douta(dout));

//Program Starts Here....
always @(posedge clk) begin

    if(rst) begin
            addr = 0;
            wrStatus = WRITE;
    end
    if(enable) begin
        case(wrStatus)
            WRITE: begin
                if(addr >= NUMELEMENTS-1) begin
                    wrStatus = DONE;
                end
                else begin
                    addr = addr + 1;
                end
            end
            DONE: begin
                addr = 0;
            end
        endcase
    end
end



endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// tx count result to uart
//
//////////////////////////////////////////////////////////////////////////////////
module timing_output(clk, timing_result, char_out);

input clk;
input [31:0] timing_result;
output [7:0] char_out;

parameter ASCII_NUM = 48;
parameter ASCII_LET = 55;

reg [3:0] char = 0;
reg [7:0] sel = 0;
reg [7:0] outReg = 0;
reg [31:0] count = 0;
wire done;

assign char_out[7:0] = outReg[7:0];
assign done = (sel == 0);

always @(posedge clk) begin

    case(sel)
        8'h0: begin
            char = count[31:28];
            sel = sel+1;
        end
        8'h1: begin
            char = count[27:24];
            sel = sel+1;
        end
        8'h2: begin
            char = count[23:20];
            sel = sel+1;
        end
        8'h3: begin
            char = count[19:16];
            sel = sel+1;
        end
        8'h4: begin
            char = count[15:12];
            sel = sel+1;
        end
        8'h5: begin
            char = count[11:8];
            sel = sel+1;
        end
        8'h6: begin
            char = count[7:4];
            sel = sel+1;
        end
        8'h7: begin
            char = count[3:0];
            sel = sel+1;
        end
        8'h8: begin
            outReg = 8'hff;
            sel = 0;
        end
        default: sel = 0;
    endcase

    if (outReg == 8'hff && sel ==0) begin
        outReg = 8'h0d;
    end
    else if(char <= 9) begin
        outReg = char + ASCII_NUM;
    end
    else begin
        outReg = char + ASCII_LET;
    end

end

always @(posedge done) begin
    count <= timing_result[31:0];
end


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// This module is design to determine the overall run time of the XCORR_Test scenario
//
// Aaron Myers
// winter 2012
//
//////////////////////////////////////////////////////////////////////////////////
module timingAnalyzer(clk, start, storing_result, result);

//define I/O
input clk;
input start;
input storing_result;
output [31:0] result;

//define element
wire timing_str;
reg [15:0] count;

//assignment statements
assign timing_str = start | storing_result;
assign result = {16'h0, count};

//program code
always @(posedge clk) begin

    if (timing_str)
        count <= count + 1;

end

endmodule
```

```
`timescale 1ns / 1ps
// Documented Verilog UART
// Copyright (C) 2010 Timothy Goddard (tim@goddard.net.nz)
// Distributed under the MIT licence.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
// THE SOFTWARE.
//
module uart(
    input clk, // The master clock for this module
    input rst, // Synchronous reset.
    output tx, // Outgoing serial line
    input transmit, // Signal to transmit
    input [7:0] tx_byte, // Byte to transmit
    output is_transmitting // Low when transmit line is idle.
    );

parameter CLOCK_DIVIDE = 109; // clock rate (50Mhz) / (baud rate (115200) * 4)

// States for the transmitting state machine.
// Constants - do not override.
parameter TX_IDLE = 0;
parameter TX_SENDING = 1;
parameter TX_DELAY_RESTART = 2;

reg [10:0] tx_clk_divider = CLOCK_DIVIDE;

reg tx_out = 1'b1;
reg [1:0] tx_state = TX_IDLE;
reg [5:0] tx_countdown;
reg [3:0] tx_bits_remaining;
reg [7:0] tx_data;

assign tx = tx_out;
assign is_transmitting = tx_state != TX_IDLE;

always @(posedge clk) begin
    if (rst) begin
```

```
        tx_state = TX_IDLE;
    end

    // The clk_divider counter counts down from
    // the CLOCK_DIVIDE constant. Whenever it
    // reaches 0, 1/16 of the bit period has elapsed.
   // Countdown timers for the receiving and transmitting
    // state machines are decremented.
    tx_clk_divider = tx_clk_divider - 1;
    if (!tx_clk_divider) begin
        tx_clk_divider = CLOCK_DIVIDE;
        tx_countdown = tx_countdown - 1;
    end


    // Transmit state machine
    case (tx_state)
        TX_IDLE: begin
            if (transmit) begin
                // If the transmit flag is raised in the idle
                // state, start transmitting the current content
                // of the tx_byte input.
                tx_data = tx_byte;
                // Send the initial, low pulse of 1 bit period
                // to signal the start, followed by the data
                tx_clk_divider = CLOCK_DIVIDE;
                tx_countdown = 4;
                tx_out = 0;
                tx_bits_remaining = 8;
                tx_state = TX_SENDING;
            end
        end
        TX_SENDING: begin
            if (!tx_countdown) begin
                if (tx_bits_remaining) begin
                    tx_bits_remaining = tx_bits_remaining - 1;
                    tx_out = tx_data[0];
                    tx_data = {1'b0, tx_data[7:1]};
                    tx_countdown = 4;
                    tx_state = TX_SENDING;
                end else begin
                    // Set delay to send out 2 stop bits.
                    tx_out = 1;
                    tx_countdown = 8;
                    tx_state = TX_DELAY_RESTART;
                end
            end
        end
        TX_DELAY_RESTART: begin
            // Wait until tx_countdown reaches the end before
            // we send another transmission. This covers the
            // "stop bit" delay.
            tx_state = tx_countdown ? TX_DELAY_RESTART : TX_IDLE;
        end
    endcase
end

endmodule
```

Appendix B: MATLAB Code

```matlab
%==========================================================================
% Testing the baud generatation code for the verilog UART module found on
% fpga4fun.com
%
% Aaron Myers
% Winter 2012
%==========================================================================
clc; clear all; close all;

%Define Variables
baud = int32(115200);
clkfrequency = int32(50e6);
baud8 = int32(8*baud);
accWidth = int32(16);
tclk = 1/double(clkfrequency);
time = 0:tclk:.0001-tclk;
clkout = zeros(1, length(time));

%Determine Increment
increment = ((baud8*2^(accWidth-7))+clkfrequency*2^-8)/(clkfrequency*2^-7);

clkAcc = int32(0);
for n = 1:length(time),
    clkAcc = clkAcc + increment;
    if(bitget(clkAcc, 17)==1)
        clkAcc = 0;
        clkout(n) = 1;
    end
end
```

```
%--------------------------------------------------------------------------
% BRO fft
%
% This scrip will execute the simulink model of the fft processor that has
% been designed to process input data in bit reversed order
%
% Aaron Myers, Winter 2012
%--------------------------------------------------------------------------
clc; clear; close all;

% Run simulation
load('tw0.mat');
load('tw1.mat');
load('tw2.mat');
load('tw3.mat');
load('inputSignal.mat');
sim('Radix2_SDF_FFT');

% Analyze fft
load('SimResult.mat');
get(result);
get(input);
out = bitrevorder(squeeze(result.Data(1024:2047)));



figure;
hold on;
plot(abs(out), 'r', 'linewidth', 3);
plot(abs(fft(squeeze(input.Data(1:1024)), 1024)), 'b');
```

```
%==========================================================================
% This file will explore the results that was provided by the fpga
% following the complex multiplier.  Results provided as a continuous
% stream of serial hex data.
%
% Aaron Myers
% Winter 2012
%==========================================================================
clc; clear all; close all;

fid = fopen('corrResult.txt', 'r');
A = fscanf(fid, '%c\n', [4, inf]);
A = A.';
fclose(fid);

A = cellstr(A);
A = reshape(A, 2, 10240)';
A_r = A(:,1);
A_i = A(:,2);

result_r = zeros(size(A,1), 1);
result_i = zeros(size(A,1), 1);
result = zeros(size(A,1), 1);

%Convert ascii values to actual hex
for n = 1:length(A),
    result_r(n) = double(typecast(uint16(hex2dec(A_r{n})), 'int16'))/2^3;
    result_i(n) = double(typecast(uint16(hex2dec(A_i{n})), 'int16'))/2^3;
end

result = complex(result_r, result_i);
```

```
load('fpgaCOEGeneration_20130108.mat', 'inputSig', 'template', 'tr_fixed');
expectedFFTResult = zeros(length(inputSig), 1);
expectedFiltResult = zeros(length(inputSig), 1);

for n = 0:9,
    expectedFFTResult(n*1024+[0:1023]+1) = fft(inputSig(n*1024+[0:1023]+1), 1024);
    expectedFiltResult(n*1024+[0:1023]+1) = expectedFFTResult(n*1024+[0:1023]+1).*templa
    expectedCorrResult(n*1024+[0:1023]+1) = ifft(expectedFiltResult(n*1024+[0:1023]+1)),
end

figure
plot(abs(result));
set(gca, 'fontsize', 14, 'xlim', [0 10240], 'ylim', [-10 1000])
legend('FPGA XCORR');

figure
hold on
plot(abs(expectedCorrResult), 'r', 'linewidth', 4);
plot(abs(result), 'linewidth', 3);
set(gca, 'fontsize', 14, 'xlim', [0 10240], 'ylim', [-10 1100]);
legend('matlab result', 'FPGA XCORR');

% figure;
% hold on;
% plot(abs(result), 'b');
% plot(abs(expectedFFTResult), 'r');
% legend('fpga FFT (16-bit resolution)', 'matlab FFT');
```

```
%-----------------------------------------------------------------------------
% FPGACorrelation
%
% This scrip test the radix2^2 SDF FFT's ability to accomplish correlation
% on an FPGA.  The FPGACorrelator model will serve as the model upon wich
% the verilog code will be written for actual implemenation
%
% Aaron Myers, Winter 2012
%-----------------------------------------------------------------------------
clc; clear; close all;

% Load the necessary variables
load('tw0.mat');
load('tw1.mat');
load('tw2.mat');
load('tw3.mat');
load('tw0_ifft.mat');
load('tw1_ifft.mat');
load('tw2_ifft.mat');
load('tw3_ifft.mat');
load('inputSignal.mat');
load('filtData.mat');

% Run the simulation
sim('FPGACorrelator');

% Analyze Corr Results
load('CorrResult.mat');
load('FFTResult.mat');
fpgaCorr = get(CorrResult);
fpgaFFT = get(FFTResult);
```

```
% Matlab's fft correlation answer
sigin = get(inputSignal);
sigin = squeeze(sigin.Data);
matlab_fft = zeros(length(sigin), 1);
matlab_result = zeros(length(sigin), 1);

for n = 0:length(sigin)/1024 - 1,
    matlab_fft(n*1024+[0:1023]+1) = abs(fft(sigin(n*1024+[0:1023]+1), 1024));
    matlab_result(n*1024+[0:1023]+1) = abs(ifft(fft(sigin(n*1024+[0:1023]+1), 1024).*fi
end

% Get Simulation Output
dstart = 4096;
fftstart = 2048;
fft_result = abs(squeeze(fpgaFFT.Data(fftstart:fftstart+10239)));
result = abs(squeeze(fpgaCorr.Data(dstart:dstart+10239)));
diff = result-matlab_result;
fft_diff = fft_result-matlab_fft;

% Plot Correlation Results
h0 = figure;
hold on
plot(abs(result), 'r', 'linewidth', 3);
plot(abs(matlab_result), 'b');
% legend('Simulink Result', 'Matlab Result');
set(gca, 'fontsize', 14);

h0b = figure;
hold on
plot(abs(result), 'r', 'linewidth', 3);
plot(abs(matlab_result), 'b');
% legend('Simulink Result', 'Matlab Result');
```

```
set(gca, 'xlim', [0 40]);

[h0_m h0_i] = inset(h0, h0b);
legend(h0_m,'Simulink Result', 'Matlab Result', 'location', 'southeast');
close(h0, h0b);

mean((diff.^2), 1)
MSE = mean((diff.^2), 1).*ones(length(diff), 1);

h1 = figure;
hold on;
bar(diff.^2, 'edgecolor', 'none');
plot(MSE, 'r--', 'linewidth', 3);
set(gca, 'ylim', [0 30], 'xlim', [0 10240], 'fontsize', 14);
legend('Squared Error', 'MSE');


% Plot FFT Results
h2 = figure;
hold on;
plot(fft_result, 'r', 'linewidth', 3);
plot(matlab_fft);
set(gca, 'xlim', [0 10240], 'ylim', [-10 200], 'fontsize', 14);


h3 = figure;
hold on;
plot(fft_result, 'r', 'linewidth', 3);
plot(matlab_fft);
set(gca, 'xlim', [0 80], 'ylim', [0 80]);

[h_m h_i] = inset(h2, h3);
```

```
legend(h_m,'Simulink FFT', 'Matlab FFT', 'location', 'northwest');
close(h2, h3);

h4 = figure;
plot(fft_diff.^2);

%Save Figures
% cd('C:\Users\Aaron\Dropbox\Thesis\Chapter4\Figures\');
% saveas(h0, 'CorrAlgorithm_vs_matlab_2', 'png');
% pause(5);
% saveas(h1, 'CorrAlgorithm_vs_matlab_2_error', 'png');
```

```
%============================================================================
% This file will explore the results that was provided by the fpga
% fft routine.  Results provided as a continuous stream of serial
% hex data.
%
% Aaron Myers
% Winter 2012
%============================================================================
clc; clear all; close all;

fid = fopen('fftCapture.txt', 'r');
A = fscanf(fid, '%c\n', [4, inf]);
A = A.';
fclose(fid);

A = cellstr(A);
A = reshape(A, 2, 10240)';
A_r = A(:,1);
A_i = A(:,2);

result_r = zeros(size(A,1), 1);
result_i = zeros(size(A,1), 1);
result = zeros(size(A,1), 1);

%Convert ascii values to actual hex
for n = 1:length(A),
    result_r(n) = double(typecast(uint16(hex2dec(A_r{n})), 'int16'))/2^8;
    result_i(n) = double(typecast(uint16(hex2dec(A_i{n})), 'int16'))/2^8;
end

result = complex(result_r, result_i);
```

```
load('fpgaCOEGeneration_20130108.mat', 'inputSig', 'template');
expectedFFTResult = zeros(length(inputSig), 1);
expectedCorrResult = zeros(length(inputSig), 1);
projectedCorr = zeros(length(inputSig), 1);

for n = 0:9,
    expectedFFTResult(n*1024+[0:1023]+1) = fft(inputSig(n*1024+[0:1023]+1), 1024);
    projectedCorr(n*1024+[0:1023]+1) = ifft(result(n*1024+[0:1023]+1).*template.');
    expectedCorrResult(n*1024+[0:1023]+1) = ifft(expectedFFTResult(n*1024+[0:1023]+1).*t
end

expectedFFTResult_fp = int32(expectedFFTResult.*2^11);

figure
hold on
plot(abs(projectedCorr), 'linewidth', 2);
% plot(abs(expectedCorrResult), 'r');
set(gca, 'fontsize', 14, 'xlim', [0 10240]);
legend('projected XCORR');

h1 = figure;
hold on;
plot(abs(result), 'b');
plot(abs(expectedFFTResult), 'r');
set(gca, 'fontsize', 14, 'xlim', [0 10240], 'ylim', [-15 200]);

h2 = figure;
hold on;
plot(abs(result), 'b', 'linewidth', 3);
plot(abs(expectedFFTResult), 'r', 'linewidth', 3);
set(gca, 'xlim', [0 40]);
```

```
[hm, hi] = inset(h1, h2)
legend(hm,'fpga FFT', 'matlab FFT', 'location', 'northwest');
```

```
%=========================================================================
% This file will explore the results that was provided by the fpga
% following the complex multiplier.  Results provided as a continuous
% stream of serial hex data.
%
% Aaron Myers
% Winter 2012
%=========================================================================
clc; clear all; close all;

fid = fopen('filtTest.txt', 'r');
A = fscanf(fid, '%c\n', [4, inf]);
A = A.';
fclose(fid);

A = cellstr(A);
A = reshape(A, 2, 10240)';
A_r = A(:,1);
A_i = A(:,2);

result_r = zeros(size(A,1), 1);
result_i = zeros(size(A,1), 1);
result = zeros(size(A,1), 1);

%Convert ascii values to actual hex
for n = 1:length(A),
    result_r(n) = double(typecast(uint16(hex2dec(A_r{n})), 'int16'))/2^4;
    result_i(n) = double(typecast(uint16(hex2dec(A_i{n})), 'int16'))/2^4;
end

result = complex(result_r, result_i);
```

```
load('fpgaCOEGeneration_20130108.mat', 'inputSig', 'template', 'tr_fixed');
expectedFFTResult = zeros(length(inputSig), 1);
expectedFiltResult = zeros(length(inputSig), 1);
projectedCorr = zeros(length(inputSig), 1);

for n = 0:9,
    expectedFFTResult(n*1024+[0:1023]+1) = fft(inputSig(n*1024+[0:1023]+1), 1024);
    projectedCorr(n*1024+[0:1023]+1) = ifft(result(n*1024+[0:1023]+1));
    expectedFiltResult(n*1024+[0:1023]+1) = expectedFFTResult(n*1024+[0:1023]+1).*templa
    expectedCorrResult(n*1024+[0:1023]+1) = ifft(expectedFiltResult(n*1024+[0:1023]+1)),
end

figure
hold on
plot(abs(expectedCorrResult), 'r', 'linewidth', 2);
plot(abs(projectedCorr));
legend('matlab result', 'projected XCORR');


% delayTestSig = [0; expectedFFTResult(1:end-1)];
% for n = 0:9,
%     delayFiltResult(n*1024+[0:1023]+1) = delayTestSig(n*1024+[0:1023]+1).*template.';
%     delayCorrResult(n*1024+[0:1023]+1) = ifft(delayFiltResult(n*1024+[0:1023]+1));
% end
%
% figure
% hold on
% plot(abs(delayCorrResult), 'r', 'linewidth', 2);
% % plot(abs(projectedCorr));
% legend('delayed result', 'projected XCORR');
```

```
%=========================================================================
%in order to test the fpga correlation algorithm the input signal and the
%template will be stored in ROM.  This should enable the assessment of the
%algorithm's performance (speed, accuracy, etc.)
%
%Aaron Myers
%Winter 2012
%=========================================================================
clc; clear all; close all;

%Initialize Variables
sum = 0;
ipname = 'inputSig.txt';
trname = 'templateReal.txt';
tiname = 'templateImag.txt';
load('fpgaCOEGeneration_20130108.mat', 'inputSig');%comment when gen new signal

%Generate a test input/template signal (double precision)
% inputSig = randn(1,1024*10);
fft_2 = fft(inputSig(1025:2048),1024).';
template = conj(fft_2); %aribtrarily picked the 2nd "pulse"
t_real = real(template);
t_imag = imag(template);

% %Plot the expected result
% result = zeros(1, length(inputSig));
% result_real = zeros(1, 1024);
% result_imag = zeros(1, 1024);
%
% for n = 0:9,
%     result_imag = int16(imag(template)).*int16(imag(fft(inputSig(n*1024+[0:1023]+1)))
%     result_real = int16(real(template)).*int16(real(fft(inputSig(n*1024+[0:1023]+1)))
%     result(n*1024+[0:1023]+1) = ifft(double(complex(result_real, result_imag))./1024)
% end
%
% figure
% plot(abs(result));

%Covert the double precision vectors to signed, 16-bit integers (assumes 1
%sign bit, 4 int bits, and 11 fraction bits).  Range => -16:15.9995
inputSig_fixed = int16(inputSig.*2^11);

%1 sign bit, 7 int bits, and 8 sign bits. Range => -128:127.9961
tr_fixed = int16(t_real.*2^8);
ti_fixed = int16(t_imag.*2^8);

%Save files as text files in hex format so that they can be made into .coe
%files for the initialization of xilinx ROM

%Input Signal
fid = fopen(ipname, 'w');

for j = 1:length(inputSig_fixed),
    fprintf(fid, '%04X\n', typecast(inputSig_fixed(j),'uint16'));
end
fclose(fid);
clear fid j;

%Real part of template
fid = fopen(trname, 'w');

for j = 1:length(tr_fixed),
    fprintf(fid, '%04X\n', typecast(tr_fixed(j),'uint16'));
end
```

```
fclose(fid);
clear fid j;

%Imaginary part of template
fid = fopen(tiname, 'w');

for j = 1:length(ti_fixed),
    fprintf(fid, '%04X\n', typecast(ti_fixed(j),'uint16'));
end
fclose(fid);
clear fid j;
```

```
%===========================================================================
%exploration of the effect of taking randn double values and transforming
%them to 16 bit fixed point notation with the value represented in two's
%complement format with four integer bits and 11 fraction bits or 7 and 8.
%
%Aaron Myers
%Winter 2012
%===========================================================================
clc; clear all; close all;

A = randn(1,1024);
a = int16(A.*2^8);
af = zeros(1, length(a));
sum = 0;
filename = 'inputSig.txt';
filename2 = 'inputSig.csv';

for n = 1:length(a),
    for i = 1:15,
        currentBit = double(bitget(a(n),i));
        sum = sum + 2^(-8+(i-1))*currentBit;
    end
    sum = sum - 2^7*double(bitget(a(n),16));
    af(n) = sum;

    sum = 0;

end

diff = A-af;

figure;
```

```
stem(diff)


figure;
hold on;
plot(abs(xcorr(A,af)), 'r', 'linewidth', 3);
plot(abs(xcorr(A,A)), 'b');
```

```
%----------------------------------------------------------------------------
% Radix-2^2 SDF IFFT
%
% This scrip will execute the simulink model of the inverse fft processor
% that has been designed to process fft input data
%
% Aaron Myers, Winter 2012
%----------------------------------------------------------------------------
clc; clear; close all;

% Initialize input signal
load('inputSignal.mat');
signal = get(input);
IFFTdata = fft(squeeze(signal.Data(1:1023)), 1024);
time = signal.Time(1:1024);
IFFTin = timeseries(IFFTdata, time);
save('IFFTin.mat', 'IFFTin', '-v7.3');

% Run simulation
load('tw0_ifft.mat');
load('tw1_ifft.mat');
load('tw2_ifft.mat');
load('tw3_ifft.mat');
load('IFFTin.mat');
sim('Radix2_SDF_IFFT');

% Analyze IFFT Return
load('IFFTout.mat');
out = get(IFFTout);
result = bitrevorder(squeeze(out.Data(1024:2047)));

plot(real(result));
```

```
%---------------------------------------------------------------------------
% Signal Generator
%
% This scrip generates the input signal into the model
%
% Aaron Myers, Winter 2012
%---------------------------------------------------------------------------
clc; clear; close all;

% Generate input signal
signal = randn(1, 10240);
time = 0:10239;
inputSignal = timeseries(signal, time);
filtData = conj(fft(signal(1025:2048), 1024));
save('inputSignal.mat', 'inputSignal', '-v7.3');
save('filtData.mat', 'filtData');
```

```
%==========================================================================
% Timing Analysis for Correlation Routines
%
% Aaron Myers
% Winter 2012
%==========================================================================
clc; clear all; close all;

%FPGA Results
fpga_corr = 10.442e-6;
fpga_corr10 = 104.42e-6;

%variables
signal = randn(10240, 1);
template = conj(fft(signal(1025:2048), 1024));
temp2 = signal(1025:2048);
result = zeros(length(signal), 1);
time1a = 0;
time2a = 0;
time1b = 0;
time2b = 0;

%Single Transform xcorr
tic;
ans1 = xcorr(temp2, temp2);
time1a = toc;

%Single Transform FFT/IFFT
tic;
ans2 = ifft(fft(signal(1025:2048), 1024).*template);
time1b = toc;
```

```
%10 Collection xcorr
tic;
ans3 = xcorr(signal, temp2);
time2a = toc;

%10 Collection FFT/IFFT
tic;
for n = 0:9,
    result(n*1024+[0:1023]+1) = ifft(fft(signal(n*1024+[0:1023]+1),1024).*template);
end
time2b = toc;

names = {'XCORR'; 'FFT/IFFT'; 'FPGA Corr'};
timing_results1 = 10^3.*[time1a, time1b, fpga_corr];
timing_results10 = 10^3.*[time2a, time2b, fpga_corr10];

h1 = figure;
bar(timing_results1);
set(gca, 'XTickLabel',names, 'XTick',1:numel(names), 'ylim', [0 40],'fontsize', 14);
ylabel('time (ms)');

h2 = figure;
bar(timing_results1);
set(gca, 'ylim', [0 .15]);
ylabel('time (ms)');

[hm, hi] = inset(h1, h2)
close(h1, h2);



figure;
bar(timing_results10);
```

```
set(gca, 'XTickLabel',names, 'XTick',1:numel(names), 'fontsize', 14);
ylabel('time (ms)');
```

```
N = 1024;

i = 0:1:(log(1024)/log(4) - 2)

a = N./(2.^(2+2.*i))

tw0 = [ones(1,N)];

tw1 = [ones(1,N/4)];

tw2 = [ones(1,N/(2^4))];

tw3 = [ones(1,N/(2^6))];

tw0_BR = [ones(1,N)];

tw1_BR = [ones(1,N/4)];

tw2_BR = [ones(1,N/(2^4))];

tw3_BR = [ones(1,N/(2^6))];



%compute the twiddle factors for stage 0

for n = a(1)+1:length(tw0),

    if n <= 2*a(1),

        v = 2*(n-a(1)-1);

    elseif (n > 2*a(1)) && (n <= 3*a(1)),

        v = (n-(2*a(1))-1);

    elseif (n > 3*a(1)) && (n <= 4*a(1)),

        v = 3*(n-(3*a(1))-1);

    end

    tw0(n) = exp(-j*2*pi*v/N);

end


%compute the twiddle factors for stage 1

for n = a(2)+1:length(tw1),

    if n <= 2*a(2),

        v = 2^3*(n-a(2)-1);

    elseif (n > 2*a(2)) && (n <= 3*a(2)),

        v = 4*(n-(2*a(2))-1);

    elseif (n > 3*a(2)) && (n <= 4*a(2)),
        v = 12*(n-(3*a(2))-1);
    end
    tw1(n) = exp(-j*2*pi*v/N);
end

%compute the twiddle factors for stage 2
for n = a(3)+1:length(tw2),
    if n <= 2*a(3),
        v = 2^5*(n-a(3)-1);
    elseif (n > 2*a(3)) && (n <= 3*a(3)),
        v = 2^4*(n-(2*a(3))-1);
    elseif (n > 3*a(3)) && (n <= 4*a(3)),
        v = 3*2^4*(n-(3*a(3))-1);
    end
    tw2(n) = exp(-j*2*pi*v/N);
end

%compute the twiddle factors for stage 3
for n = a(4)+1:length(tw3),
    if n <= 2*a(4),
        v = 2^7*(n-a(4)-1);
    elseif (n > 2*a(4)) && (n <= 3*a(4)),
        v = 2^6*(n-(2*a(4))-1);
    elseif (n > 3*a(4)) && (n <= 4*a(4)),
        v = 3*2^6*(n-(3*a(4))-1);
    end
    tw3(n) = exp(-j*2*pi*v/N);
end

tw0_BR = bitrevorder(tw0);
tw1_BR = repmat(bitrevorder(tw1), 1,4);
tw2_BR = repmat(bitrevorder(tw2), 1,16);
```

```
tw3_BR = repmat(bitrevorder(tw3), 1,64);
tw0_ifft = conj(tw0);
tw1_ifft = conj(tw1);
tw2_ifft = conj(tw2);
tw3_ifft = conj(tw3);

save('tw0.mat', 'tw0');
save('tw1.mat', 'tw1');
save('tw2.mat', 'tw2');
save('tw3.mat', 'tw3');
save('tw0_BR.mat', 'tw0_BR');
save('tw1_BR.mat', 'tw1_BR');
save('tw2_BR.mat', 'tw2_BR');
save('tw3_BR.mat', 'tw3_BR');
save('tw0_ifft.mat', 'tw0_ifft');
save('tw1_ifft.mat', 'tw1_ifft');
save('tw2_ifft.mat', 'tw2_ifft');
save('tw3_ifft.mat', 'tw3_ifft');
```

Appendix C: Compressed Sampling Theory

UWB radars and communication systems have always been constrained by the sampling rates of available ADCs. To further complicate things, the sampling of UWB waveforms creates large amount of data that is difficult to process in a timely manner. This miniaturization effort compounds both these problems by reducing the available sampling bandwidth and computing horse power. In order to combat these issues, compressive sensing (CS) will be explored.

Reconstruction of the received signal is a vital aspect of any radar or communication system. Conventional systems are able to accomplish this reconstruction by sampling according to the Nyquist-Shannon theorem. This theorem requires that the incoming signal be sampled at a rate slightly above twice the highest frequency content. If, however, the signal of interest is sparse in a given domain, the information may be contained in a much smaller effective bandwidth. Using CS, it may be possible to fully reconstruct the received signal with far fewer samples than required by traditional methods. Furthermore, CS is a particularly interesting proposition for RNR as "CS exploits the fact that many natural signals are sparse or compressible in the sense that they have concise representations in the proper basis ψ" [3].

To describe the process of CS the following mathematical development is often presented: Consider a discretely sampled, one-dimensional signal $x = [x[1], \cdots, x[N]]$ that can be represented as an Nx1 vector in N-dimensional space. Let $\Psi = [\psi_1, \cdots, \psi_n]$. Ψ is an NxN dimensional matrix where the ψ_i's are Nx1 basis vectors. Now x can be represented as

$$x = \sum_{i=0}^{N} s_i \psi_i, \; or \; x = \mathbf{\Psi s} \; [29]. \tag{C.1}$$

Now, K will be defined as the number of non-zero coefficient in x. For x to be considered sparse in ψ, K needs to be much smaller than N. If the signal, x, is sparse, then only M

samples of x will be collected, where $K < M << N$. The result is an M-length observation vector, y that can be expressed as

$$y = \mathbf{\Phi x} \quad [40]. \tag{C.2}$$

Substituting Equation (C.1) into Equation (C.2) one arrives at

$$y = \mathbf{\Phi \Psi s} = \mathbf{\Theta s}. \tag{C.3}$$

According to Candes and Wakin, sparsity in and of itself is not a sufficient condition for CS. The product of ϕ and ψ, Θ must be incoherent [3]. To ensure this condition is upheld Θ must satisfy the restricted isometric property (RIP) [9]. If ϕ is chosen to be a independent identically distributed Gaussian matrix, then Θ has been shown to satisfy the RIP with high probability for many orthonormal bases: spikes, sinusoids, wavelets, Gabor functions, and curvelets [40].

Once the sparsity and incoherence properties have been satisfied, the reconstruction of the signal of interest, x, is accomplished by solving the l1-norm minimization problem:

$$\hat{s} = arg\ min\ ||s||_1\ s.t.\ y = \mathbf{\Phi \Psi s} \quad [29]. \tag{C.4}$$

Equation (C.4) is one of several proposed techniques to recover sparse signals, but is the one that is most commonly used.

Bibliography

[1] Axelsson, Sune R.J. "Generalized Ambiguity Functions for Ultra Wide Band Random Waveforms". *Radar Symposium, 2006. IRS 2006. International*, 1 –4. may 2006.

[2] Burrus, C.S. and T.W. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley and Sons, 1985.

[3] Candes, E.J. and M.B. Wakin. "An Introduction To Compressive Sampling". *Signal Processing Magazine, IEEE*, 25(2):21 –30, march 2008. ISSN 1053-5888.

[4] Chi, J.C. and S. Chen. "An efficient FFT Twiddle Factor Generator". *Proc. European Signal Processing Conference*, 1533–1536.

[5] 1st Class Michael Guillory, Sgt. "Up, Up and Away", November 2006. URL http://www.army.mil/-images/2006/11/22/1024/army.mil-2006-11-22-114612.jpg. ...and away it goes, on an aerial reconnaissance mission for Iraqi and U.S. Soldiers on the ground.

[6] Collins, Peter J. and John A. Priestly III. "An Investigation of the Trade-offs Between Electronic Protection and Processing Efficiency in a Multistatic Noise Radar Network". *Waveform Diversity Conference, 2012*. 2012.

[7] Cooley, James W. and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". 19(90).

[8] Dereniak, E.L. and G.D. Boreman. *Infrared Detectors and Systems*. John Wiley and Sons, Inc., 1996.

[9] Donoho, D. "Compressed Sensing". *IEEE Trans. on Information Theory*, 52(4):1289–1306, April 2006.

[10] Garmatyuk, D. S. and R. M. Narayanan. "ECCM capabilities of an ultrawideband bandlimited random noise imaging radar". *Aerospace and Electronic Systems, IEEE Transactions on*, 38(4):1243–1255, 2002. ID: 1.

[11] Garrido, M., J. Grajal, and O. Gustafsson. "Optimum Circuits for Bit Reversal". *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 58(10):657 –661, oct. 2011. ISSN 1549-7747.

[12] Grant, M.P., G.R. Cooper, and A.K. Kamal. "A class of noise radar systems". *Proceedings of the IEEE*, 51(7):1060 – 1061, july 1963. ISSN 0018-9219.

[13] Guosui, Liu, Gu Hong, and Su Weimin. "Development of random signal radars". *Aerospace and Electronic Systems, IEEE Transactions on*, 35(3):770–777, 1999. ID: 1.

[14] He, Shousheng and M. Torkelson. “A new approach to pipeline FFT processor”. *Parallel Processing Symposium, 1996., Proceedings of IPPS ’96, The 10th International*, 766 –770. apr 1996.

[15] Horton, B.M. “Noise-Modulated Distance Measuring Systems”. *Proceedings of the IRE*, 47(5):821 –828, may 1959. ISSN 0096-8390.

[16] Lai, Chieh-Ping and R.M. Narayanan. “Through Wall Surveillance Using Ultrawideband Random Noise Radar”. 2006.

[17] Lai, Chieh-Ping and R.M. Narayanan. “Ultrawideband Random Noise Radar Design for Through-Wall Surveillance”. *Aerospace and Electronic Systems, IEEE Transactions on*, 46(4):1716 –1730, oct. 2010. ISSN 0018-9251.

[18] Lievsay, James R. *Simultaneous Range/Velocity Detection with an Ultra-Wideband Random Noise Radar through Fully Digital Cross-Correlation in the Time Domain*. Master’s thesis, Air Force Institute of Technology, Wrigh-Patterson AFB, OH, 2011.

[19] Maximum Integrated Products. *8-Bit, 2.2Gsps ADC with Track/Hold Amplifier and 1:4 Demultiplexed LVDS Outputs*. URL http://datasheets.maximintegrated.com/en/ds/MAX109.pdf.

[20] Maximum Integrated Products. *Application Note 800: Design a Low-Jitter Clock for High-Speed Data Converters*. URL http://www.maximintegrated.com/an800.

[21] McGillem, C.D., G.T. Cooper, and W.B. Waltaman. “An Experimental Random Signal Radar”. *Proceedings of the National Electronics Conference*, 409–411. October 1967.

[22] Narayanan, Ram M., Robert D. Mueller, and Robert D. Palmer. “Random noise radar interferometry”. 75–82, 1996. URL +http://dx.doi.org/10.1117/12.257243.

[23] Narayanan, R.M. and M. Dawood. “Doppler estimation using a coherent ultrawideband random noise radar”. *Antennas and Propagation, IEEE Transactions on*, 48(6):868 –878, jun 2000. ISSN 0018-926X.

[24] Narayanan, R.M., Y. Xu, P.D. Hoffmeyer, and J.O. Curtis. “Design and performance of a polarimetric random noise radar for detection of shallow buried targets”. *Proc. SPIE Meeting on Detection Techn. Mines, Orlando*, volume 2496, 20–30. 1995.

[25] Nelms, Matthew. *Development and Evaluation of a Multistatic Ultrawideband Random Noise Radar*. Master’s thesis, Air Force Institute of Technology, Wrigh-Patterson AFB, OH, 2010.

[26] Osborn, Leon, Jeffrey Brummond, Robert Hart, Mohsen Zarean, and Steven Conger. “Clarus Concept of Operations”. FHWA-JPO-05-072. 2005.

[27] Priestly, John A. III. *AFIT NoNET Enhancements: Software Model Development and Optimization of Signal Processing Architecture*. Master's thesis, Air Force Institute of Technology, 2011.

[28] Rabiner, Lawrence R. and Bernard Gold. *Theory and Application Of Digital Signal Processing*. Prentice-Hall, Inc., 1975.

[29] Rawat, Ankit Singh and Kartik Venkat. "Compressed Sensing and its Applications in UWB Communication Systems", Oct 2010. Term paper written for EENG 670, Stanford University.

[30] Richards, Mark A., James A. Scheer, and William A. Holm (editors). *Principles of Modern Radar: Basic Principles*. SciTech Publishing, Inc., 2010.

[31] Saeed, Ahmed, M. Elbably, G. Abdelfadeel, and M.I. Eladawy. "Efficient FPGA Implementation of FFT/IFFT Processor". 3.

[32] Smit, J. A. "Radar - An experimental noise radar system". *AGARD Conference Proceedings*, 39.1–39.7. 1970.

[33] Sule, Ambarish Mukund. "Design of Pipeline Fast Fourier Transform Processors Using 3 Dimensional Integrated Circuit Technology". URL http://www.lib.ncsu.edu/resolver/1840.16/3879.

[34] Thayaparan, T. and C. Wernik. *Noise Radar Technology Basics*. Technical report, DEFENCE RESEARCH AND DEVELOPMENT CANADA OTTAWA (ONTARIO);, Dec-2006.

[35] Theron, I.P., E.K. Walton, and S. Gunawan. "Compact range radar cross-section measurements using a noise radar". *Antennas and Propagation, IEEE Transactions on*, 46(9):1285–1288, 1998.

[36] Thorson, T. Joel. *Simulataneous Range-Velocity Processing and SNR Analysis of AFIT's Random Noise Radar*. Master's thesis, Air Force Institute of Technology, Wrigh-Patterson AFB, OH, 2012.

[37] Thorson, T. Joel and Geoffrey A. Akers. "Investigating the Use of a Binary ADC for Simultaneous Range and Velocity Processing in a Random Noise Radar", 2010. Note.

[38] Thorson, T. Joel and Geoffrey A. Akers. "Near Real-Time Simultaneous Range and Velocity Processing in a Random Noise Radar", 2011. Note.

[39] Walton, EK, IP Theron, S. Gunawan, and L. Cai. "Moving vehicle range profiles measured using a noise radar". *Antennas and Propagation Society International Symposium, 1997. IEEE., 1997 Digest*, volume 4, 2597–2600. IEEE, 1997.

[40] Wu, Ji, Qilian Liang, Zheng Zhou, Xiaorong Wu, and Baoju Zhang. "Compressive sensing for sense-through-wall UWB noise radar signal". *Communications and Networking in China (CHINACOM), 2011 6th International ICST Conference on*, 979 –983. aug. 2011.

[41] Xilinx, Inc. *ISE Help*. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/isehelp_start.htm.

[42] Xilinx, Inc. *ISE In-Depth Tutorial*. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/ise_tutorial_ug695.pdf.

[43] Xilinx, Inc. *Virtex-5 FPGA ML555 Development Kit For PCI and PCI Express Designs, User Guide*. URL http://www.xilinx.com/support/documentation/boards_and_kits/ug201.pdf.

[44] Zatrepalek, Reg. "Using FPGAs to Solve Tough DSP Design Challenges". *Xcell Journal*.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 21-03-2013	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* Oct 2011 - Mar 2013

4. TITLE AND SUBTITLE The Miniaturization of the AFIT Random Noise Radar	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Myers, Aaron T., Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB, OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-13-M-37
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Advanced Navigation Technology Center Attn: Dr. John F. Raquet 2950 Hobson Way WPAFB, OH 45433-7765	**10. SPONSOR/MONITOR'S ACRONYM(S)** ANT
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Advances in technology and signal processing techniques have opened the door to using an UWB random noise waveform for radar imaging. This unique, low probability of intercept waveform has piqued the interest of the U.S. DoD as well as law enforcement and intelligence agencies alike. While AFIT's noise radar has made significant progress, the current architecture needs to be redesigned to meet the space constraints and power limitations of an aerial platform. This research effort is AFIT's first attempt at RNR miniaturization and centers on two primary objectives: 1) identifying a signal processor that is compact, energy efficient, and capable of performing the demanding signal processing routines and 2) developing a high-speed correlation algorithm that is suited for the target hardware. A correlation routine was chosen as the design goal because of its importance to the noise radar's ability to estimate the presence of a return signal. Furthermore, it is a computationally intensive process that was used to determine the feasibility of the processing component. To determine the performance of the proposed algorithm, results from simulation and experiments involving representative hardware were compared to the current system. Post-implementation reports of the FPGA-based correlator indicated zero timing failures, less than a Watt of power consumption, and a 44% utilization of the Virtex-5's logic resources.

15. SUBJECT TERMS

Random Noise Radar, FPGA Correlation, Noise Radar Miniaturization

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Peter J. Collins (AFIT/ENG)
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	169	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636 x7526